Computer Science and Information Systems 20(1):307–327 https://doi.org/10.2298/CSIS210908058P

Complete Formal Verification of the PSTM Transaction Scheduler*

Miroslav Popovic[1], Marko Popovic[1], Branislav Kordic[1], and Huibiao Zhu[2]

[1] University of Novi Sad, Faculty of Technical Sciences, Trg D. Obradovica 6,
21000 Novi Sad, Serbia
{miroslav.popovic, marko.popovic, branislav.kordic}@rt-rk.uns.ac.rs
[2] East China Normal University, Shanghai Key
Laboratory of Trustworthy Computing,
Shanghai 200062, China
hbzhu@sei.ecnu.edu.cn

Abstract. State of the art formal verification is based on formal methods and its goal is proving given correctness properties. For example, a PSTM scheduler was modeled in CSP in order to prove deadlock-freeness and starvation-freeness. However, as this paper shows, using solely formal methods is not sufficient. Therefore, in this paper we propose a complete formal verification of trustworthy software, which jointly uses formal verification and formal model testing. As an example, we first test the previous CSP model of PSTM transaction scheduler by comparing the model checker PAT results with the manually derived expected results, for the given test workloads. Next, according to the results of this testing, we correct and extend the CSP model. Finally, using PAT results for the new CSP model, we analyze the performance of the PSTM online transaction scheduling algorithms from the perspective of the relative speedup.

Keywords: Formal Verification, Process Algebra, Transaction Scheduling, Python, Software Transactional Memory.

1. Introduction

As contemporary society is becoming increasingly dependent on software, which is ubiquitously used in everyday life, software verification is gaining paramount importance for our society and the environment. State of the art formal verification based on model checking is performed in two steps: (i) constructing a formal model of a given safety critical software, and (ii) proving that this formal model satisfies a given set of correctness properties, which consists of safety and liveness properties. For example, a PSTM (Python Software Transactional Memory) transaction scheduler was recently modeled (see [1]) in process algebra Communicating Sequential Processes (CSP) [2], in order to automatically prove the subject's deadlock-freeness (a safety property) and starvation-freeness (a liveness property) by the model checker PAT (Process Analysis Toolkit) [3].

* A preliminary version of this paper appears in the Proceedings of the 7th Conference on the Engineering of Computer Based Systems (ECBS), article no. 10, pages 1-10, Novi Sad, Serbia, May 2021 [25].

However, as this paper shows, conducting traditional formal verification in two steps (described above) as was, for example, done in [1], [4], [5], and [6] is not sufficient. As will be elaborated in more detail in Section 1.1 (related work), the main problem with the traditional two-steps formal verification is that the formal model constructed in the first step is not directly tested. Therefore, possible formal model shortcomings may not be discovered, and consequently, they may compromise the formal verification results. As a solution to this problem, in this paper, we propose a method for a complete formal verification of trustworthy software, which jointly uses formal verification and formal model testing. Our method is based on the iterative procedure with the following steps (the procedure inputs comprise the initial formal model and the manually derived expected results):

1. Test the formal model by using the model checker and the expected results.
2. If the results are not as expected, correct the formal model and return to step 1.
3. Make the final report.

In the paper, we demonstrate our method using an example in which we applied the complete formal verification on the PSTM transaction scheduler. Within the example, we: (i) tested the previous CSP model of PSTM transaction scheduler by comparing the model checker PAT results with the manually derived expected results, for the given test workloads, (ii), according to the testing results, we corrected and extended the CSP model in each iteration (see the last two paragraphs in Section 1.1 for the history of all the corrections that were made in more iterations), and (iii) using PAT results for the final CSP model (henceforth called “the new CSP model”), we analyzed the performance of PSTM online transaction scheduling algorithms from the perspective of the relative speedup.

The rest of the paper is organized as follows. Section 1.1 presents closely related work, Section 2 presents the testing of the previous CSP model, Section 3 presents the new CSP model, Section 4 presents the performance analysis of the four PSTM online transaction scheduling algorithms, and Section 5 presents the paper conclusions.

1.1. Related Work

A brief overview of the most closely related research presented in this section covers formal verification and its testing, PSTM, and PSTM transaction scheduler formal verification chronology.

The formal verification process used in this paper is based on model checking. “Model checking is a technique for automatic verification of software and reactive system, and it consists of verifying some properties of the model of the system”, see [7]. We selected three recent papers in order to illustrate formal verification state of the art [4], [5], and [6].

The paper [4] was motivated by the importance of the discovery and control service of an IoT system based on the Chord protocol, and the obvious fact that errors in concurrent systems are difficult to reproduce and find using solely program testing. The authors manually proved the correctness of the Chord protocol using the logic of time and knowledge with the respect to the set of possible executions (that maintain ring topology while the nodes can freely join or leave). The given proof was not automatically verified in one of the formal proof assistants (e.g., Coq, Isabelle/HOL), and the authors only mention this as a possible challenge for their future work.

The paper [5] addresses the issues of safety-critical software verification and testing that are key requirements for achieving DO-178C and DO-331 regulatory compliance for airborne systems. The verification is performed by the symbolic model checker MCMAS+ that uses OBDDs. To validate their model, the engineers need to perform review and tracing activities. Review means fixing syntax errors, whereas tracing means checking model behaviour along all the possible traces within the complete model's state space. Both activities are conducted manually, so they are time-consuming and error-prone. Moreover, it seems that validation is not based on theoretically expected results, so the engineers are left to handle it according to their experience and intuition.

The paper [6] presents an approach for specifying, verifying, and deploying time-constrained business processes (BPs) in a mono-cloud, multi-edge context. At design-time, four stages take place: (i) specification in Business Process Model and Notation (BPMN), (ii) placement of tasks and data on cloud or edge, (iii) transformation from BPMN model to Timed Petri-Nets (TPN) model, and (iv) verification whether TPN model has any time violations (this is done automatically using a model checker like TINA). The main disadvantage of this approach is that the initial BPMN model is not checked. Additionally, the engineers: (i) do not derive some kind of theoretically expected results of the placement, and (ii) they do the placement manually as a series of trial-and-error attempts.

Testing as defined in the Cambridge Dictionary is "the process of using or trying something to see if it works, is suitable, obeys the rules, etc." [8]. In this paper, we test the formal model, or more precisely we test the formal verification process itself, in order to cross-check whether it produces the expected theoretical results. To the best of our knowledge, this is the first paper that advocates and demonstrates such an approach to the complete formal verification. So, the testing performed in this paper should not be confused with software testing, which is considered to be woefully inadequate for detecting errors in highly concurrent designs [7].

Transactional memory (TM) was conceived as an extension of a cache-coherence protocol that supports transactions executed on multicores, which operate on shared variables (called t-variables, or t-vars) [9], [10]. Software TM (STM) appeared as a TM implemented in software [11]. Python STM (PSTM) [12] is a general purpose STM for Python, which is applicable in a wide range of application-specific domains, from computational chemistry simulations [13], to data science, to IoT-based systems such as smart homes, vehicles, and cities.

PSTM was formally verified in three complementary papers. The authors of the first paper [14] constructed a formal model as a network of timed automata [15] representing: linear and cyclic transactions, the queue used by the remote procure call mechanism, and the transactional memory itself. Using the model checker UPPAAL [16], they automatically proved the following three properties: safety (atomicity), liveness (in a set of concurrent transactions, one must get committed), and termination (the cyclic transactions must eventually get committed).

The authors of the second paper [17] constructed a formal model as a group of CSP processes representing: a transaction, PSTM API, PSTM server, and PSTM dictionary. Using the model checker PAT, they automatically proved the following three properties:

deadlock freeness, ACI (atomicity, consistency, and isolation), and optimism (essentially the same as the liveness in [14]).

The authors of the third paper [18] constructed a formal PSTM push/pull semantic model as the mapping of operations within the PSTM's generic transactional algorithm to the four relevant push/pull rules: PULL, APPLY, PUSH, and CMT (the general push/pull semantic model is defined in [19]). They manually proved that the model satisfies correctness criteria for the relevant push/pull rules, and that therefore PSTM satisfies serializability (i.e. sequential consistency).

Generally, STM transactions are easy to program (as a simple sequential code that is executed atomically), they cannot deadlock (since one of the concurrent transactions gets committed), and they provide great performance for low concurrency workloads (since they are executed speculatively, i.e. without the overhead incurred by locks). However, in case of high concurrency workloads, the system performance may be degraded, because many transactions may get aborted and re-executed, or even worse, some of the transactions may be starved. Therefore, various transaction schedulers (or contention managers) were introduced in order to sustain good performance even for high concurrency workloads, e.g. [20], [21], [22].

PSTM transaction scheduler architecture and the four online scheduling algorithms, named Round Robin (RR), Execution Time Load Balancing (ETLB), Avoid Conflicts (AC), and Advanced Avoid Conflicts (AAC), were developed with the main goal to minimize the makespan (the total execution time) and consequently to maximize the throughput (the number of transactions per second) for an arbitrary workload [23], [24]. These algorithms were developed hierarchically, from the simplest RR to the most advanced AAC, and they were compared from the perspectives of time complexity, quality of theoretical initial schedules, and the experimentally measured speedup over RR and the number of aborts.

The theoretical schedules in [24] were manually derived for three test workloads: CFW (Conflict Free Workload), RDW (Read Dominated Workload), and WDW (Write Dominated Workload), which were previously used for the experimental evaluation in [23]. The experimental results and the theoretical results are well aligned, and this fact validates both the theoretical analysis in [24] and the algorithms' implementations in Python and their experimental evaluation in [23].

The authors of the paper [1] constructed the formal model of the PSTM scheduler architecture and the first three online transaction scheduling algorithms (RR, ETLB, and AC) from [23] as a group of CSP processes representing: an application, the scheduler input queue, the scheduler, the worker input queues, the workers, and the processes formalizing RR, ETLB, and AC algorithms. Using the model checker PAT, they automatically proved deadlock and starvation freeness properties and analyzed algorithms performance from the perspective of makespan, relative speedup, number of aborts, and throughput. However, instead of using test workloads from [23] and [24], they introduced three simplified test workloads: CFW, CW-1 (Conflict Workload 1), and CW-2 (Conflict Workload 2), so a direct comparison of their results with the results of [23] and [24] was not possible. Moreover, the formal verification in [1] was made with some shortcomings, which we discovered and remedied in [25].

In [25], we tested the CSP# model of the PSTM transaction scheduler introduced in [1] by comparing the model checker PAT results with the manually derived expected results, for the test workloads defined in [1], and we discovered six shortcomings. Next, we extended the CSP# model in order to eliminate the discovered shortcomings. Finally, using PAT, we automatically analyzed the performance of the PSTM transaction scheduling algorithms from the perspective of makespan and throughput.

In this paper, we conduct a complete formal verification of the PSTM transaction scheduler for the three test workloads similar to those used in [23] and [24], in order to be able to compare the PAT's results with the results in [23] and [24]. More precisely, we first derived complete theoretical schedules for these workloads and tested the CSP# model from [25] with them. We discovered two additional shortcomings of the CSP# model from [25], and we extended the model accordingly. Next, using PAT, we analyzed the performance of the PSTM transaction scheduling algorithms from the perspective of the relative speedup. Finally, we compared the PATs results with the previous experimental results in [23].

2. Testing

This section presents the testing of the formal models developed in [1] and [25]. The main goal of the testing was to check whether the formal verification results are aligned with the theoretical results. The next three subsections present the testing method, theoretical schedules for the given test workloads, and the testing findings.

2.1. Testing Method

The testing method is based on the analysis of theoretical schedules, which are expected to be produced by the subject online transaction scheduling algorithms for the given test workloads. The method comprises the following steps:

- derive theoretical schedules;
- calculate the makespan and the number of aborts;
- compare the results of the previous step with the model checker PAT's results.

In addition to the test workloads used in [1] and [25], in this paper, we introduce the new test workload, named: CF (Conflict Free), RD (Read Dominated), and WD (Write Dominated) workloads, which are similar to the three test workloads (CFW, RDW, and WDW) used in [24]. The main difference between the former and the latter is that the former have 6 transactions, whereas the latter have 12 transactions. The second difference is that each transaction in the CF workload writes to a single t-variable, whereas each transaction in the CFW in [24] is a money transfer (MT) transaction i.e. it reads and writes two t-variables.

The new test workloads (CF, RD, and WD) are modeled in CSP# as arrays of six transactions [T_0, …, T_5]. There are three kinds of transactions: M, R, and W transactions. M transaction writes to a single t-variable from the set of t-variables {A, B, C, D, E, F}, R transaction sequentially reads all the t-variables, and W transaction sequentially writes to all the t-variables. The duration of the M transaction is 10 time units, whereas the durations of the R and W transactions are 40 time units each (these durations are the same as in [24]). The CF workload is a series of M transactions operating of different t-variables, the RD workload is the interleaving of R transactions and M transactions, and the WD workload is the interleaving of W and M transactions.

2.2. Theoretical Schedules

As cores in a multicore processor use the same clock, a multicore processor is a synchronous system and therefore transactions assigned to separate workers' cores execute synchronously in parallel. The experiments conducted in [23] confirm this fact, and it was accepted as a postulate when formalizing PSTM transaction scheduler architecture and deriving the theoretical schedules for the given number of workers and given test workloads.

We derived the theoretical schedules for the workloads introduced in [1] in the preliminary version of this paper [25] (see them there). Here we consider the new test workloads (CF, RD, and WD). In order to save space, we derive the theoretical schedules for the PSTM scheduler architecture only with two workers, because these simple schedules are easy to comprehend and interpret even by readers not too familiar with this topic. The theoretical schedules for three test workloads CF, RD, and WD are shown in Fig. 1, Fig. 2, and Fig. 3, respectively.

In each figure, the queue with input transactions is shown at its top, where each transaction is labeled with its index (i.e. its ID in the CSP# model) and its type (M, R, or W). The expected schedules for individual online transaction scheduling algorithms are shown below the input queue, where the two workers' queues are labeled W_0 and W_1, respectively.

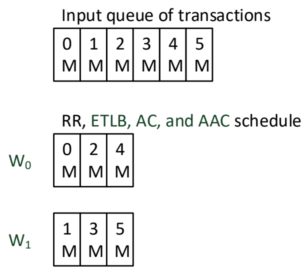

Fig. 1. The theoretical schedules for CF workload

Fig. 1 shows the expected schedule for the CF workload, which is conflict free, because all the transactions operate on different t-variables. Since there are two workers, the RR algorithm works by modulo two, so it assigns even transactions to W_0 and odd ones to W_1 in its first scheduling iteration. This schedule is executed without aborts (because there are no conflicts) and therefore this schedule happens to be the complete schedule with the makespan equal to 30 (3 x 10) and the number of aborts equal to 0.

Because all the transactions have the same duration, the ETLB algorithm behaves as the RR algorithm, and because there are no potential conflicts among transactions (since they operate on different t-variables), the AC and the AAC algorithms behave as the

ETLB algorithms, i.e. the RR algorithm. So, all the algorithms produce the same schedule shown in Fig. 1.

Figure 2 shows the expected schedule for the RD workload, which has 9 potential conflicts – these are the conflicts between each R transaction (T_0, T_2, and T_4) and each M transaction (T_1, T_3, and T_5) because R transactions sequentially read all the t-variables and M transactions (T_1, T_3, and T_5) write to a single t-variable (B, D, and F, resp.). Since there are two workers, the RR algorithm works by modulo two, so it assigns even transactions to W_0 and odd to W_1 in its first scheduling iteration. In this initial schedule, T_0 (assigned to W_0) is in conflict with all the M transactions (assigned to W_1). Because all the M transactions end before T_0, they all get committed, whereas T_0 gets aborted, rescheduled and successfully re-executed in the second iteration. So the complete schedule has the makespan equal to 160 (4 x 40), and the number of aborts is equal to 1.

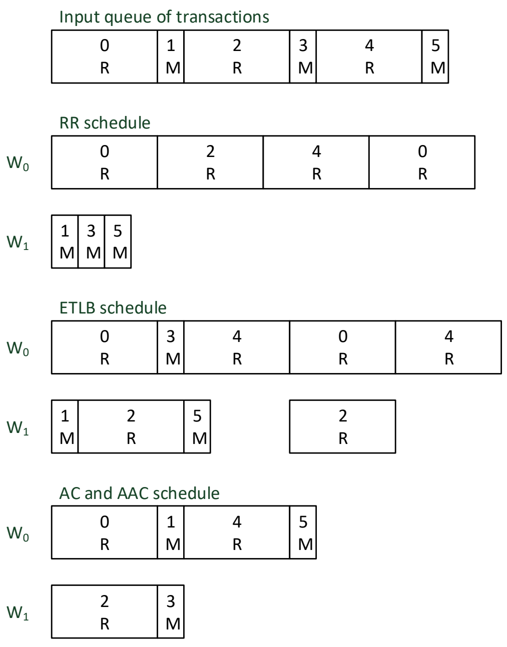

Fig. 2. The theoretical schedules for RD workload

The ETLB algorithm performs load balancing by making the following sequence of assignments in its first iteration (see Fig. 2, ETLB schedule): T_0 to W_0, T_1 to W_1, T_2 to W_1, T_3 to W_0, T_4 to W_0, and T_5 to W_1. However, this initial schedule is worse than the one produced by the RR algorithm, because it is executed with the following 3 conflicts causing 3 aborts: (i) T_0 and T_1 are in conflict, and T_1 is faster, so T_0 gets aborted, (ii) T_2 and T_3 and in conflict, and they end simultaneously, so in the worst case T_2 gets aborted (this is the worst case because T_2's duration is greater than T_3's) and (iii) T_4 and T_5 are in conflict, and T_5 is faster, so T_4 gets aborted. In the second iteration, the ETLB makes the following series of assignments (see Fig. 2): T_0 to W_0, T_2 to W_1, and T_4 to W_0. This schedule is executed without aborts because it is conflict free. So the complete schedule has the makespan equal to 170 (4 x 40 + 10), and the number of aborts is equal to 3.

The AC and the AAC algorithms do better than the ETLB algorithm because they are aware of the potential conflicts. Actually, for this particular workload, the AC algorithm produces the same result as the optimal AAC algorithm using a simple heuristic (schedule the next transaction to the least loaded worker if that does not cause a conflict). Both algorithms make the following sequence of assignments (see Fig. 2, AC and AAC schedule): T_0 to W_0, T_1 to W_0 (because of the potential conflict between T_0 and T_1), T_2 to W_1, T_3 to W_1 (because there is no conflict between T_2 and T_3), T_4 to W_0, and T_5 to W_0 (because of the potential conflict between T_4 and T_5). This initial schedule is conflict free, and therefore constitutes the complete schedule with the makespan equal to 100 (2 x 40 + 2 x 10), and the number of aborts equal to 0.

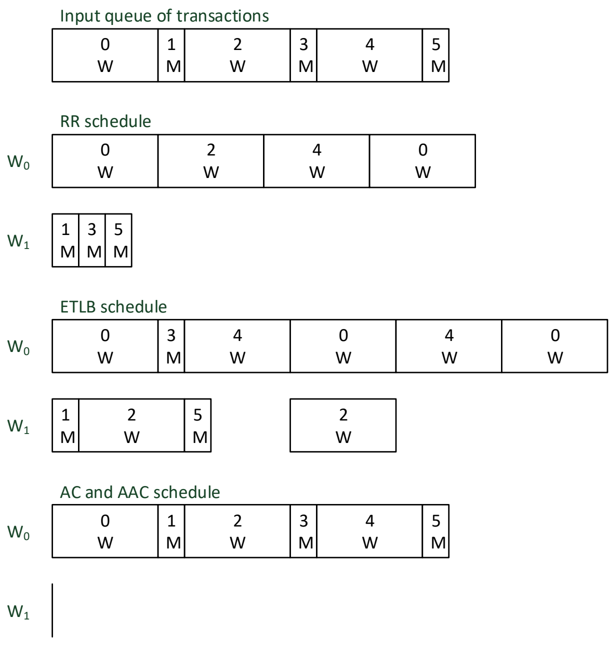

Fig. 3. The theoretical schedules for WD workload

Figure 3 shows the expected schedule for the WD workload, which has potential conflicts among all the transactions. The RR algorithm works by modulo two, so it assigns even transactions to W_0 and odd to W_1 in its first scheduling iteration. In this initial schedule, T_0 (assigned to W_0) is in conflict with all the M transactions (assigned to W_1). Because all the M transactions end before T_0, they all get committed, whereas T_0 gets aborted, rescheduled and successfully re-executed in the second iteration. So the complete schedule has the makespan equal to 160 (4 x 40), and the number of aborts is equal to 1. This result is practically the same as that for the RD workload (see Fig. 2).

The ETLB algorithm performs load balancing by making the following sequence of assignments in its first iteration (see Fig. 3, ETLB schedule): T_0 to W_0, T_1 to W_1, T_2 to W_1, T_3 to W_0, T_4 to W_0, and T_5 to W_1. This initial schedule is practically the same as for the RD workload (see Fig. 2), and for the same reasons T_0, T_2, and T_4 get aborted (in the worst case) and rescheduled in the second iteration. However, unlike the case for the RD workload, T_0 and T_2 are now in conflict, so one of them (say T_0) gets aborted and re-executed in the third iteration. So, the complete schedule has the makespan equal to 210 (5 x 40 + 1 x 10) and the number of aborts equal to 4.

The AC and AAC algorithms serialize all the transactions (because of the conflicts among them) by assigning them all to W_0, whereas W_1 stays idle (see Fig. 3, AC and AAC schedule). So, this initial conflict-free schedule constitutes the complete schedule with the makespan equal to 150 (3 x 40 + 3 x 10) and the number of aborts equal to 0.

Table 1 summarizes the expected theoretical results for the makespan (*ms*) and the number of aborts (*na*) for the number of workers (*Index*) equal to 2 (derived above), as well as for the *Index* equal to 3, and 4 (which are derived analogously).

Table 1. The expected theoretical results for CF, RD, and WD workloads

Load & Alg.		*Index* = 2		*Index* = 3		*Index* = 4	
Load	Alg.	*ms*	*na*	*ms*	*na*	*ms*	*na*
CF	RR	30	0	20	0	20	0
	ETLB	30	0	20	0	20	0
	AC	30	0	20	0	20	0
	AAC	30	0	20	0	20	0
RD	RR	160	1	90	3	120	2
	ETLB	170	3	100	3	90	3
	AC	100	0	60	0	70	0
	AAC	100	0	50	0	50	0
WD	RR	160	1	180	7	160	3
	ETLB	210	4	180	6	170	6
	AC	150	0	150	0	150	0
	AAC	150	0	150	0	150	0

2.3. Testing Findings

Formal model testing was organized in two phases. The object of the first phase was the initial CSP# model from [1], whereas the object of the second phase was the extended CSP# model from [25]. The first phase was conducted in [25] using the three test workloads defined in [1], (CFW, CW-1, and CW-2), whereas the second phase was conducted in this paper using the three test workloads very similar to those used in [23] and [24], namely CF, RD, and WD workloads.

Table 2 summarizes the testing findings. In the first testing phase, we discovered items 1, 5, and 6 in Table 2, as well as the initial shortcomings in items 2-4. Next, in the second testing phase, we discovered items 7 and 8, as well as additional shortcomings in items 2-4.

By comparing the expected values of makespan and the values in [1], we found that, by an oversight, the time used (TU) for the model checking by the model checker PAT was interpreted as equal to the makespan. Next, the fact that the expected values for the number of aborts and the values in [1] were different, indicated that the real schedules from [1] are different from the expected schedules derived in [25]. A rather tedious reconstruction of the real schedules from the model checker PAT's log files confirmed this indication.

Table 2. The summary of testing findings

No.	Finding
1	PAT's time used (TU) reported as the makespan
2	The macro isConflict shortcomings
3	Pessimistic concurrency control
4	Asynchronous transactions' execution
5	The number of workers fixed to *Index* = 2
6	AAC algorithm not supported
7	Read operations and the star convention not supported
8	The macros findmin and findmax shortcomings

The three main root causes of the discrepancies between the real and the expected schedules (the findings 2-4 in Table 2) are the following: (i) the macro isConflict, which checks whether there are conflicts between the next transaction to be scheduled *x* and any already scheduled transaction *i*, included the case *i* = *x*, (ii) the CSP# model from [1] uses the pessimistic concurrency control, whereas the real PSTM uses the optimistic concurrency control (see Sec. 2-1 in [10]), and (iii) the workers in the CSP# model from [1] execute transactions asynchronously, and therefore this model violates the postulate of the synchronous transaction execution introduced in Section 2.2.

By a quick inspection of the model from [1], we saw that the number of workers is fixed to *Index* = 2 and that the AAC algorithm is not supported (items 5-6 in Table 2). In the second testing phase, we needed to model RD and WD workloads, and at that time we discovered that the model in [25] does not support read operations and the star convention by which * means all the t-variables (item 7 in Table 2). While fixing item 7, we found additional shortcomings related to items 2-4 and we discovered that in the case when the array *load* has equal elements, both macro findmin and macro findmax return the index of the last of them, whereas in the theoretical schedules we took the index of the first such element (item 8).

3. The New Model

This section presents the new CSP# model, which evolved from the previous model presented in [25]. The next subsections present the three most important parts of the model related to (i) conflict detection, (ii) optimistic concurrency control, and (iii) synchronous transactions' execution. This organization was made according to the findings 2-4 and 7 in Section 2.3 (other extensions are skipped because of the space limits).

3.1. Conflict Detection

The macro isConflict sets the variable *IsConflict* to 0 if the new transaction *x* starting at *t1* and ending at *t2* is not in conflict with the already scheduled transactions; otherwise it sets *IsConflict* to 1, see Algorithm 1. Initially, it sets the variables *i* and *IsConflict* to 0 (lines 4-5; *i* is the index of the *i*-th transaction). Next, it repeats the loop while *i* is less than TNum (the number of transactions) and there is no conflict (lines 6-18).

Within the loop, it performs a series of nested if statements, which is equivalent to a single if statement with a conjunction of all the conditions, in order to check whether the transaction *i* is scheduled (line 7), and the transactions *i* and *x* overlap in time (lines 8-9), and *i* is not *x*, and the transactions *i* and *x* operate on the same t-variable or one of them operates all the t-variables as indicated by the constant Star (line 10), and the transactions *i* or *x* write to the t-variable(s) (line 11), and if yes it sets *isConflict* to 1.

Algorithm 1. The macro isConflict

```
01: var IsConflict = 0;
02: #define isConflict(x, t1, t2)
03: {
04:   var i = 0;
05:   IsConflict = 0;
06:   while(i<TNum && IsConflict!=1) {
07:     if(T_isScheduled[i]==1) {
08:       if((T_StartTime[i]<t2 && t2<=T_EndTime[i])||
09:       (t1<T_EndTime[i] && T_EndTime[i]<=t2)) {
10:         if((i!=x)&&(T_Var[i]==T_Var[x]||T_Var[i]==Star||T_Var[x]==Star)) {
11:           if(T_VarOp[i]==Wtvar || T_VarOp[x]==Wtvar) {
12:             IsConflict = 1;
13:           }
14:         }
15:       }
16:     }
17:     i++;
18:   }
19: };
```

3.2. Optimistic Concurrency Control

The PSTM's optimistic concurrency control model is shown in Algorithm 2. This model is much simpler than the complete PSTM models developed in [14] and [17], and it was made as such, in order to keep the model state space exploration fast and feasible.

The array *T_VarVer* (line 1) models the PSTM dictionary, and its elements store the versions of the respective t-variables (TvarNum is the number of t-variables). Since the number of transactions in a workload is equal to TNum, we bounded the number of versions for each t-variable by TNum, in order to reduce the state space to be explored (if TNum transactions update a single t-variable, its final version would be TNum-1). The number of a t-variable versions is effectively bounded by counting them with modulo TNum (line 13 for single t-variable updates and line 18 for the updates using the star convention).

In order to support updates using the star convention (i.e. updates of all the t-variables), we define the *star version* (i.e. the version of all the t-variables) as the sum of the versions of all the t-variables. The macro calciver (called in line 6) stores its result in the variable *iver* (line 2), and the result is either the star version if the argument *key* is equal to Star, or the version *T_VarVer*[*key*] if *key* is not equal to Star.

The process *Pstm* models the PSTM itself (lines 3-29). The worker processes send their messages to *Pstm* over the channel *worker2pstm*, whereas *Pstm* sends its replies to the worker process W_i via the channel *pstm2worker*[*i*].

Algorithm 2. The PSTM's optimistic concurrency control model

```
01: var T_VarVer[TvarNum]:{0..TNum-1} = [0(TvarNum)];
02: var iver:{0..TvarNum*TNum} = 0;

03: Pstm() =
04:   worker2pstm?i.req.op.key.ver->
05:   atomic {
06:     {call(calciver, key)}->
07:     if(req == GetVars) {
08:       pstm2worker[i]!iver->Pstm()
09:     } else { // commitVars
10:       if(ver == iver) { // T_VarVer[key] replaced with iver
11:         if(op == Wtvar) {
12:           if(key != Star) {
13:             {T_VarVer[key] = (T_VarVer[key]+1)%TNum}->
14:             pstm2worker[i]!1->Pstm()
15:           } else {
16:             {
17:               var ii = 0;
18:               while(ii<TvarNum){T_VarVer[ii]=(T_VarVer[ii]+1)%TNum;ii++}
19:             }->
20:             pstm2worker[i]!1->Pstm()
21:           }
22:         } else {
23:           pstm2worker[i]!1->Pstm()
24:         }
25:       } else {
26:         pstm2worker[i]!0->Pstm()
27:       }
28:     }
29:   };
```

The compound messages sent by W_i to *Pstm*, over the channel *worker2pstm*, have the format *i.req.op.key.ver*, where *i* is the worker's index, *req* is the type of request (the existing types of requests are: GetVars and CommitVars, which correspond to the PSTM API functions getVars and commitVars, respectively), *op* is the type of operation (the existing types of operations are: Wtvar and Rtvar, which correspond to the write and read operations, respectively), *key* is the index of the t-variable or Star, and *ver* is the version of the t-variable or the star version.

The replies sent from *Pstm* to W_i, over the channel *pstm2worker*[*i*], have a single element whose semantics depend on the type of the request: for GetVars request, the reply is the t-variable' version, whereas for CommitVars request, the reply is either 0 or 1 whether the transaction gets aborted or successfully committed, respectively.

After receiving the message *i.req.op.key.ver* (line 4) the process *Pstm* atomically (line 5) serves the request as follows. It calls the macro calciver (line 6) to set the *iver* for the given *key* (which may be an index of a t-variable or Star). Then it checks the type of the request (line 7). In the case of the GetVars request (line 8), it returns *iver* (which may be the version of a t-variable or the star version). Alternatively, in the case of the CommitVars request, *Pstm* checks whether the version *ver* from the input message is equal to *iver* (line 10). If they are equal, *Pstm* checks the type of *op* (line 11).

If *op* is equal to Wtvar (line 11), *Pstm* compares *key* with Star (line 12). If *key* is not equal to Star, *Pstm* increments the current version of the t-variable key (line 13), whereas if *key* is equal to Star, *Pstm* increments current versions of all the t-variables (line 18), and in both cases *Pstm* sends the reply 1 signaling successful commit (line 20). If *op* is not equal to Wtvar (i.e. *op* is the read operation), *Pstm* sends the reply 1 signaling successful commit (line 23). If *ver* is not equal to *iver*, *Pstm* sends the reply 0 signaling abort (line 26).

As defined above, *Pstm* provides optimistic concurrency control by servicing two types of requests made by synchronous concurrent workers executing transactions. Each transaction starts with the GetVars request (to get local copies of specified t-variables), proceeds with some data processing (on local copies), and ends with the CommitVars request (to update the specified shared t-variables).

3.3. Synchronous Transactions' Execution

The worker's behavior model is shown in Algorithm 3 (excluding unimportant parts). The worker W_i initially behaves as the process *Worker*(*i*) (line 1). After receiving the signal READY from the scheduler, it behaves as the process *Worker_1*(*i*) (line 2).

The process *Worker_1*(*i*) is an iterative process (line 3). In each iteration, it checks the input queue *Queue*[*i*] (line 4). If there are no transactions in the input queue, *Worker_1*(*i*) executes the process *Worker_2*(*i*), sends the signal done to the scheduler, and continues behaving as the process *Worker*(*i*) (line 5). Alternatively, if there is a transaction in the input queue (line 6), *Worker_1*(*i*) dequeues the transaction, estimates the duration of the transaction, sets the variables related to the transaction (these steps are not shown in Algorithm 3), and executes the process *Working*(*i, currentT*[*i*]), where *currentT*[*i*] is the index of the current transaction executed by W_i (line 7).

By the definition of the parallel composition operator ||, all the parallel processes must simultaneously engage in their *common events* (i.e. the events in the intersection of their alphabets) [2]. In Algorithm 3, all the workers synchronize using the so-called *lock-step synchronization*, i.e. they engage in their common event *tick* simultaneously. Therefore, all the workers must engage in the same number of ticks, *nt*, in each scheduling iteration. In order to calculate *nt*, let nt_i be the total load plus the number of transactions allocated to W_i (the number of transactions is added because starting each transaction requires one *tick*). Then, *nt* is the maximal nt_i, nt_m (nt_i for the worker $i = m$), for $i = 0, \ldots,$ Index – 1:

$$nt_i = load_i + \text{count}(Queue_i)\ . \qquad (\mathbf{1})$$

$$nt = nt_m = max_i\ nt_i\ . \qquad (\mathbf{2})$$

Next, let it_i be the number of idle ticks that must be executed by W_i after it processed all the transactions from its input queue, $Queue_i$:

$$it_i = nt - nt_i\ . \qquad (\mathbf{3})$$

The processes *Working*(*i, txn*) and *Working_1*(*i, txn*) model the behavior of W_i processing its current transaction *txn*. The former models the start of the transaction, whereas the latter models the rest of the transaction.

Algorithm 3. The worker's behavior

```
01: Worker(i) =
02:   comSW[i]?READY->Worker_1(i);

03: Worker_1(i) =
04:   if(Queue[i].Count() == 0) {
05:     Worker_2(i); output!done -> Worker(i)
06:   } else {
        …
07:     Working(i, currentT[i]);
        …

08: Worker_2(i) =
09:   if(idleTicks[i] > 0) {
10:     tick -> tau{idleTicks[i]--} -> Worker_2(i)
11:   } else {Skip};

12: Working(i, txn) =
13:   tick ->
14:   worker2pstm!i.GetVars.0.currentT_Var[i].0 ->
15:   pstm2worker[i]?tvarver ->
16:   {currentT_VarVer[i] = tvarver; workertime[i]++} ->
17:   Working_1(i, txn);

18: Working_1(i, txn) =
19:   tick ->
20:   if(workertime[i] < currentT_Time[i]) {
21:     working{workertime[i]++} ->
22:     Working_1(i, txn)
23:   } else {
24:     worker2pstm!i.CommitVars.T_VarOp[txn].currentT_Var[i].currentT_VarVer[i] ->
25:     pstm2worker[i]?resp ->
26:     {currentT_Cmt[i] = resp; workertime[i] = 0} ->
27:     Skip
28:   };
```

The process *Working*(*i*, *txn*) (lines 12-17): (i) sends the GetVars request to the process *Pstm* (line 14) and receives the value of *tvarver* (which is either the version of the t-variable *key* if *key* is not Star or the star version if *key* is equal to Star) in the reply from *Pstm* (line 15), and (ii) stores *tvarver* into *currentT_VarVer*[*i*] and increments its working time by updating *workertime*[*i*] (line 16).

The process *Working_1*(*i*, *txn*) (lines 18-28) checks whether it has to do more processing (line 20), and if yes, it increments its working time (line 21). At the end of the transaction (line 23), it: (i) sends the CommitVars request to the process *Pstm* (line 24) and receives the *Pstm*'s reply *resp*, which is 1 if the transaction got successfully committed, otherwise it is 0 (line 25), and (ii) stores *resp* into *currentT_Cmt*[*i*] and resets the working time (for the next scheduling iteration) by clearing *workertime*[*i*] (line 26).

After the process *Working_1*(*i*, *txn*) terminates, the process *Worker_1*(*i*) resumes at line 5, where it executes the process *Worker_2*(*i*) (lines 8-11). The latter checks whether the number of its idle ticks *idleTicks*[*i*] is greater than zero (line 9), and if yes, decrements *idleTicks*[*i*] (line 10); otherwise it terminates (line 11).

4. Formal Verification

This section presents the formal verification for the new CSP# model. The next two sections present the verification results and the performance analysis.

4.1. Verification Results

First, we define the following three system conditions, which are used in the assertions:

– the condition *Done* requires that *snum* is equal to TNum (where *snum* is the number of the successfully executed transactions), which means that all the transactions have been successfully executed;
– the condition *MaxNA* requires that *na* is nonnegative and that Done is satisfied;
– the condition *MaxMS* requires that *ms* is nonnegative and that *Done* is satisfied.

Next, we introduce the five assertions that were checked for each workload defined in the previous paper [25] (CFW, CW-1, and CW-2) and in this paper (CF, RD, and WD workloads) and each version of the system, where the version of the system is defined by the given number of workers (Index) and the given online transaction scheduling algorithm (RR, ETLB, AC, and AAC). The first assertion corresponds to a safety property, and the other assertions correspond to liveness properties.

These five assertions are defined as follows:

– the system is deadlock-free;
– the system reaches a state satisfying the condition *Done*;
– all the system's evolution paths satisfy the CSP# LTL formula []<>*comSA*.complete, which means that always eventually ([]<>) the signal complete is sent from the scheduler to the application, over the channel *comSA*;
– the system reaches a state satisfying the condition *MaxNA* over a path that maximizes *na* (this is achieved by using the clause "with max(*na*)", which instructs PAT to search and report the maximal value of *na*);
– the system reaches a state satisfying the condition *MaxMS* over a path that maximizes the expression (*ms*+*na*) (this is achieved by using the clause "with max(*ms*+*na*)").

Using PAT, we checked these five assertions for each of the six workloads (CFW, CW-1, CW-2, CF, RD, and WD workloads) and for each of the system versions. There are system versions with 2, 3, and 4 workers (i.e. Index = 2, 3, 4), and for each of the 4 online transaction scheduling algorithms (RR, ETLB, AC, and AAC), i.e. there are 12 system versions (3 x 4 = 12) in total. All the 360 assertions (6 x 12 x 5 = 360) that we checked were found to be valid (i.e. satisfied).

The third assertion for the case with 4 workers, WD workload, and ETLB algorithm was the most time-consuming to validate. The verification statistics reported by PAT for this case are the following: 14386432 visited states, 46722717 passed transitions, 526 s of used time, and 6950446 KB of used memory (on a PC with 16GB DDR4 memory and CPU i7-8750H 2.2 GHz with turbo bust to 4.1 GHz).

We also manually checked the values for *na* and *ms* reported by the last two assertions, and they matched the expected results. The expected results for the workloads defined in this paper are given in Table 1, whereas the expected results for the workloads defined in [25] are given in Table 1 and Table 3 in [25].

4.2. Performance Analysis

In this section, we: (i) introduce the necessary definitions, (ii) analyze the performance of the PSTM online transaction scheduling algorithms from the perspective of the relative speedup using the theoretical results (confirmed by PAT) from [25] and this paper, and (iii) we validate the theoretical results from this paper with the experimental results from [23].

First, we define the *number of independent transactions* within a workload, *L*, which we use to quantitatively characterize the level of parallelism for a given *L*.

Definition 1. The number of independent transactions, *nit*, is the max number of transactions in *L* that could be scheduled online (i.e. without changing the transactions arrival order), in parallel, on an infinite number of processors, without a conflict.

For example, the values of *nit* for WD, RD, and CF workloads, are 1, 3, and 6, respectively. So, WD has the lowest level of parallelism, and CF has the highest.

Next, we define the *relative speedup* of an algorithm A over the algorithm RR.

Definition 2. The relative speedup, *s*, of an algorithm A over the algorithm RR is defined as the ratio $ms_{\mathrm{RR}}/ms_{\mathrm{A}}$, where ms_{RR} and ms_{A} are the makespans for the algorithms RR and A, respectively.

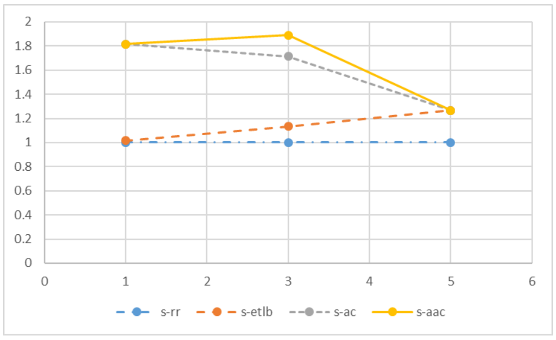

Fig. 4. The average relative speedup for the theoretical results from [25]

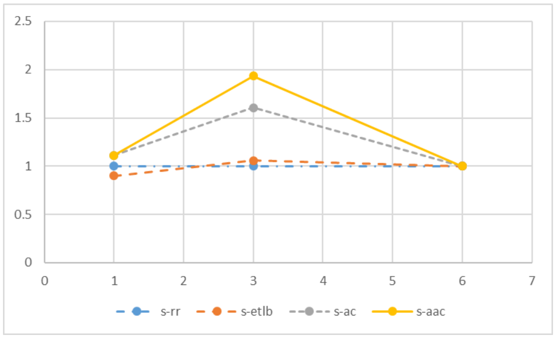

Fig. 5. The average relative speedup for the theoretical results in this paper

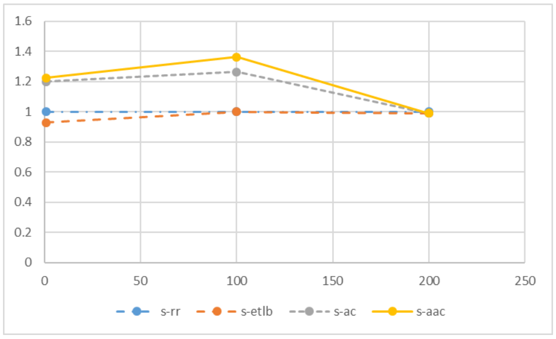

Fig. 6. The average relative speedup for the experimental results from [23]

Using data from [25], we calculated the average relative speedup for each algorithm (RR, ETLB, AC, and AAC) and each workload (CW-2, CW-1, and CFW), which are characterized by their *nit* values (1, 3, and 5); the average is calculated over the values of relative speedup for Index = 2, 3, and 4 (i.e. 2, 3, and 4 workers).

The values of the average *s* for RR, ETLB, AC, and AAC algorithms are illustrated in Fig. 4 with the curves denoted as *s-rr*, *s-etlb*, *s-ac*, and *s-aac*, respectively.

Similarly, using data from this paper, we calculated the average relative speedup for each algorithm (RR, ETLB, AC, and AAC) and each workload (WD, RD, and CF workloads), which are characterized by their *nit* values (1, 3, and 6); the average is again calculated over the values of relative speedup for Index = 2, 3, and 4. The values of the average *s* for RR, ETLB, AC, and AAC algorithms are illustrated in Fig. 5 with the curves again denoted as *s-rr*, *s-etlb*, *s-ac*, and *s-aac*, respectively.

After analyzing the data and the shape of the curves in Fig. 4 and Fig. 5, we may conclude that according to their performance, in terms of the average relative speedup, the PSTM online transaction scheduling algorithms can generally be ranked as follows: (i) AAC is the best, (ii) AC is worse than AAC, (iii) ETLB is worse than AC, and (iv) RR is the worst. There is a single exception to the general conclusion: in the case of the WD workload, ETLB is worse than RR, see Fig. 5. This exception is not unexpected, because, for some workloads (like the WD workload), the simple RR algorithm may outperform the ETLB algorithm that sometimes may be too greedy.

Next, we compare the theoretical results in this paper (illustrated in Fig. 5) with the experimental results from [23]. It is important to realize that these results are not quantitatively comparable, because of the following reasons: (i) the workloads from [23] have 200 transactions each (whereas the workloads in this paper have 6 transactions each), (ii) the duration of transactions in the workloads from [23] are: M transaction takes 0.65 ms whereas R and W transactions take 45 ms each (whereas the duration of transactions in this paper are: M takes 10 time units whereas R and W take 40 time units), (iii) the experiments in [23] were made only for the two cases: with 2 and 3 workers, and (iv) the experiments in [23] were made on a PC with 4 cores, consequently some schedules for the case with 3 workers were compromised by the local OS, since 4 cores were not sufficient to host 3 workers and the OS processes.

On the other hand, we had an intuition that these theoretical and experimental results should be qualitatively comparable, because of the following reasons: (i) the workloads from [23] and from this paper have the same rather simple patterns, which when scheduled on a small number of workers (like in [23]) should yield schedules having rather small periods, and (ii) if we take the time unit to be one ms, at least the durations of R and W transactions would be the same.

Therefore, using the data from [23], we calculated the average relative speedup for each algorithm and each workload (namely WDW, RDW, and CFW, which are characterized by their *nit* values: 1, 100, and 200, respectively); the average is calculated over the values of relative speedup for Index = 2 and 3 (2 and 3 workers). The values of the average *s* for RR, ETLB, AC, and AAC algorithms are illustrated in Fig. 6 with the curves again denoted as *s-rr*, *s-etlb*, *s-ac*, and *s-aac*, respectively.

After analyzing the shapes of the curves in Fig. 5 and in Fig. 6, we may conclude that the theoretical results in this paper are qualitatively well aligned with the experimental results from [23].

5. Conclusions

Modern society is becoming strongly dependent on the pervasive use of software in everyday life, and therefore software verification is becoming extremely important. Traditionally, formal methods have been seen as a key for the successful design of safety critical systems. However, in this research, we learned that using solely formal methods, like CSP, is not sufficient. As a solution, we proposed the method for the complete formal verification of trustworthy software, which jointly uses formal verification and formal model testing.

In the paper, we applied this method and conducted the complete verification of the PSTM online transaction scheduler and the accompanying scheduling algorithms, through the iterative procedure of testing and correcting/extending the initial CSP model. The final result of this iterative procedure is called the new CSP model. Using this new CSP model, we analyzed the performance of the PSTM online transaction scheduling algorithms from the perspective of the relative speedup, and got the results that were as expected and well aligned with the previous research [23], [24], and [25].

The main difficulty that we faced in this research was to make a realistic model that is simple enough to be checkable on an off-the-shelf PC that was at our disposal. This difficulty caused the main limitations of the presented example: (i) the limited number of workers (up to 4), (ii) the limited number of test workloads (6 altogether), (iii) the limited number of transactions in a workload (up to 6), (iv) fixed transactions' durations, and (v) the transactions either operate on a single t-variable or on all the t-variables (using the star convention). In our future work we plan: (i) to address these limitations, and (ii) to apply the complete formal verification on other software architectures.

Acknowledgement. This work was partially supported by: (i) the Ministry of Education, Science and Technology Development of Republic of Serbia under Grant 451-03-68/2020-14/200156, and (ii) the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (Grant No. 22510750100).

References

1. Xu, C., Wu, X., Zhu, H., Popovic, M.: Modeling and Verifying Transaction Scheduling for Software Transactional Memory using CSP. In Proceedings of the 13th Theoretical Aspects of Software Engineering Symposium. IEEE, Guilin, China, 240-247. (2019)
2. Hoare, C.A.R.: Communicating Sequential Processes. Prentice/Hall International, New Jersey, USA. (1985)
3. Si, Y., Sun, J., Liu, Y., Dong J. S., Pang, J., Zhang, S. J., Yang, X.: Model checking with fairness assumptions using PAT. Frontiers of Computer Science, Vol. 8, No. 1, 1-16. (2014)
4. Marinković, B., Ognjanović, Z., Glavan, P., Kos, A., Umek, A.: Correctness of the Chord Protocol. Computer Science and Information Systems, Vol. 17, No. 1, 141-160. (2020)
5. Elqortobi, M., El-Khouly, W., Rahj, Amine R., Bentahar, J., Dssouli, R.: Verification and Testing of Safety-Critical Airborne Systems: a Model-based Methodology. Computer Science and Information Systems, Vol. 17, No. 1, 271-292. (2020)
6. Cheikhrouhou, S., Kallel, S., Guidara, I., Maamar, Z.: Business Process Specification, Verification, and Deployment in a Mono-Cloud, Multi-Edge Context. Computer Science and Information Systems, Vol. 17, No. 1, 293-313. (2020)
7. Berard, B., Bidoit, J. M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, Ph., McKenzie, P.: Systems and Software Verification. Springer, Berlin, Germany. (1999)
8. Testing. Cambridge online dictionary. [Online]. Available: https://dictionary.cambridge.org/dictionary/english/testing (current September 2021)
9. Herlihy, M., Moss, J. E. B.: Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture. ACM, San Diego, CA, USA, 289-300. (1993)
10. Harris, T., Larus, J. R., Rajwar, R.: Transactional Memory, 2nd edition. Morgan and Claypool, San Rafael, CA, USA. (2010)
11. Shavit, N., Touitou, D.: Software transactional memory. In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing. ACM, Ottawa, Ontario, Canada, 204-213. (1995)
12. Popovic, M., Kordic, B.: PSTM: Python software transactional memory. In Proceedings of the 22nd Telecommunications Forum (TELFOR). IEEE, Belgrade, Serbia, 1106-1109. (2014)
13. Kordic, B., Popovic, M., Popovic, M., Goldstein, M., Amitay, M., Dayan, D.: An Evolutionary Computational System Architecture Based on a Software Transactional

Memory. Revue Roumaine des Sciences Techniques. Ser. Electrotechnique et Energetique, Vol. 61, No. 1, 47-52. (2021)
14. Kordic, B., Popovic, Ghilezan, M., S.: Formal Verification of Python Software Transactional Memory Based on Timed Automata. Acta Polytechnica Hungarica, Journal of Applied Sciences, Vol. 16, No. 7, 197-216. (2019)
15. Alur, R., Dill, D. L.: A theory of timed automata. Theoretical Computer Science, Vol. 126, No. 2, 183-235. (1994)
16. Behrmann, G., David, A., Larsen, K. G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.): Formal Methods for the Design of Real-Time Systems. Lecture Notes in Computer Science, Vol. 3185. Springer-Verlag, Berlin Heidelberg New York, 200-236. (2004)
17. Liu, A., Zhu, H., Popovic, M., Xiang, S., Zhang, L.: Formal Analysis and Verification of the PSTM Architecture Using CSP. Journal of Systems and Software, Vol. 165, 1–14. (2020)
18. Popovic, M., Popovic, M., Ghilezan, S., Kordic, B.: Formal Verification of Local and Distributed Python Software Transactional Memories. Revue Roumaine des Sciences Techniques. Ser. Electrotechnique et Energetique, Vol. 64, No. 4, pp. 423–428. (2019)
19. Koskinen, E., Parkinson, M.: The Push/Pull Model of Transactions. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. Morgan Kaufmann, Portland, Oregon, USA, 186-195. (2015)
20. Yoo R.M., Lee, H.-H.S.: Adaptive transaction scheduling for transactional memory systems. In Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures. ACM, Munich, Germany, 169–178. (2008)
21. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.): High Performance Embedded Architectures and Compilers. HiPEAC 2009. Lecture Notes in Computer Science, Vol. 5409. Springer, Berlin, Heidelberg, 4-18. (2009)
22. Dolev, S., Hendler, D., Suissa, A.: CAR-STM: scheduling based collision avoidance and resolution for software transactional memory. In Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing. ACM, Toronto, Ontario, Canada, 125-134. (2008)
23. Popovic, M., Kordic, B., Popovic, M., Basicevic, I.: Advanced Algorithm for Scheduling TM Transactions with Conflict Avoidance. In Proceedings of the 25th Telecommunications Forum (TELFOR). IEEE, Belgrade, Serbia, 844-847. (2017)
24. Popovic, M., Kordic, B., Popovic, M., Basicevic, I.: Online Algorithms for Scheduling Transactions on Python Software Transactional Memory. Serbian Journal of Electrical Engineering, Vol. 16, No. 1, 85-104. (2019)
25. Popovic, M., Popovic, M., Kordic, B., Zhu, H.: PSTM Transaction Scheduler Verification Based on CSP and Testing. In Proceedings of the 7th Conference on the Engineering of Computer Based Systems. ACM, Novi Sad, Serbia, Article No. 10, 1-10. (2021)

Miroslav Popovic received his Dipl. Eng., M.Sc., and Ph.D. degrees from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 1984, 1988 and 1990, respectively. He is a Full Professor at the University of Novi Sad from 2002. Currently he is giving courses on parallel programming, real-time systems programming, and inter-computer communications and computer networks. In the past, he has supervised many real-world projects for the industry, mostly in real-time and embedded systems. His research interests are engineering of computer-based systems, intelligent distributed systems, and security. He has authored or co-authored about 30 peer-reviewed journal papers, more than 120 conference papers, and the book Communication protocol engineering, Second Edition (CRC Press, Taylor & Francis Group, 2018).

Marko Popovic received his B.Sc., M.Sc., and PhD degrees from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 2015, 2017, and 2020, respectively. Currently he is Scientific Researcher affiliated with the RT-RK Institute of Computer Based Systems, Novi Sad, Serbia. His research interests are in the areas of engineering of computer-based systems and intelligent distributed systems. He has authored or co-authored more than 15 scientific papers.

Branislav Kordic received B.Sc, M.Sc, and PhD degrees from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 2012, 2013, and 2020, respectively. He was a teaching assistant at the Faculty of Technical Sciences, University of Novi Sad. Currently he works as a professional software engineer. His domains of interest are real-time systems and systems for parallel and distributed computing.

Huibiao Zhu is currently a professor in East China Normal University, Shanghai. He earned his Ph.D. degree in formal methods from London South Bank University, London, in 2005. During these years, he has studied various semantics and their linking theories for Verilog, SystemC, web services and probability system. He was the Chinese PI of the Sino-Danish Basic Research Center IDEA4CPS.

Received: September 08, 2021; Accepted: May 11, 2022.