Analysis by proof and verification of an abstract algorithm for distributed irregular tree processing

Astrid Kiehn Sriram Kailasam
School of Computing and Electrical Engineering
IIT Mandi
Mandi (H.P.) - 175005, India
astrid@iitmandi.ac.in, sriramk@iitmandi.ac.in

***Abstract*—In the domain of irregular tree processing, two level load-balancing strategies are most commonly used. Static load balancing is performed initially based on task estimates while dynamic load balancing is performed during run-time to deal with skewed load conditions. A particular variant of this strategy based on a master-worker architecture has been implemented for concept mining in the area of Formal Concept Analysis, [1]. It has been equipped with an additional component for fault tolerance in [2].**

In this paper we propose an abstract algorithm for irregular tree processing generalising the strategy used in [1] for concept mining. We prove deadlock-freedom and termination correctness. We investigate this algorithm further by creating PROMELA models of increasing level of abstraction to be able to analyse it with the SPIN model checker. For the most abstract and compact model we verify that there is no loss or addition of work, deadlock-freedom, termination, and the ability of workers to act as stealers and as donors in the same run. Finally, this model is extended by the features to adjust to fail-stop errors of workers guided by the implementation of [2]. While the earlier properties remain valid (modified wrt. failure of workers), adjustment to a loss of workers is also verified.

***Index Terms*—Verification, Load-Balancing, Work-Stealing Algorithm, Irregular Tree Processing**

I. Introduction

Irregular tree processing is characterized by a highly skewed structure of the computation tree. Distributing this tree evenly on a cluster of machines is challenging as the tree is generated dynamically, and the number of nodes in the sub-trees is highly uneven due to branching and pruning operations. These problems are found in several domains such as simulation/modeling, AI, discrete optimization, etc. [1], [3]–[5]. A good and recent introduction to the field can be found in [6].

In irregular tree processing, initially, there is only the root of the tree; the child nodes are generated as part of the processing. When child nodes are generated, they temporarily are leaf nodes with less work compared to their parent nodes. When a leaf node is processed it either unfolds and becomes the parent of new child nodes or it is pruned. As unfolding and pruning is not predictable, a static distribution of the child nodes among various machines causes load imbalance. Hence, dynamic load balancing (DLB) strategies are used. These strategies can be broadly divided into receiver-initiated (work stealing) and sender initiated (work sharing) schemes. Machines act as so-called workers which can process, steal or donate work items. In the work stealing scheme, a worker without work steals work from some donor machine, whereas in the work sharing scheme a worker assigns some of its work to other workers. Generally, work stealing strategies are more communication efficient than work sharing ones as the communication is initiated only when a worker becomes idle.

Work stealing is implemented either in master-worker architecture or peer-to-peer architecture. In master-worker, the master enables stealing of tasks. In peer-to-peer architecture, each worker maintains a view of all other workers and requests work from its neighbors (locality sensitive) or from some random worker [7]. The main challenges in distributed irregular tree processing are keeping the communication costs low, selecting the worker which is most appropriate for stealing or donating work, optimizing the size of work item that can be stolen, and determining termination of the entire system. In the master-based scheme the main strategy employed to tackle communication costs is that workers send their updates to the master only when their estimated load becomes half. In peer-to-peer schemes, each worker maintains its own estimate.

Over the last two decades, we see a large number of scalable systems built using master-worker architecture (e.g. MapReduce [8], Apache Spark [9], etc.). This architecture facilitates better co-ordination and control amongst workers compared to peer-to-peer architecture.

In this paper we consider a master-based system. Master-worker schemes can be classified into master-heavy and master-light schemes based on the amount of work performed by the master. In master-heavy scheme, the master maintains the list of unexplored tasks and assigns them on requests of idle workers. These schemes do not scale to large number of workers as the master becomes a bottleneck [10]. In contrast, in the master-light schemes, the master only maintains meta-data about the workers. The master maintains the list of potential donors ordered by their estimated workload. When an idle worker sends a request to the master, the master immediately returns the most loaded worker thereby facilitating efficient donor lookup. The actual exchange of work is performed between the donor and the requester. These schemes have better scalability as the tasks are maintained locally at the worker's end [1], [11]. The size estimate of the sub-trees assigned to a worker helps to avoid small task stealing.

The master-light scheme of the implementation described in

[1] has been shown to effectively deal with the computation skew of irregular tree processing. In this paper we extract the strategy employed in [1] to be applicable for irregular tree processing in general. In the implementation each worker uses a double ended queue (deque) for storing work items (deques are also used in other implementations, see [12]). The work items are kept in decreasing order of their estimates. For exploration (processing), the worker pops a work item from the bottom of the deque (low estimate end), and – if it is not consumed – generates its child items and pushes them to the bottom in the decreasing order of the estimates. This way of exploring work items bounds the total size of the deque (during execution) to be equal to the maximum degree of the computation tree. The worker donates work items from the top of the deque (high estimate end). The item at the top is the largest work item, with its estimate about half of the total queue. Thus, a worker can offload half its load by donating a single work item, thereby reducing the communication overhead. The master maintains estimates of the work available at each worker and enables constant time lookup of donors. The overall estimate kept by the master may be inaccurate occasionally due to delayed updating of the worker. However, it only affects the performance not correctness of the algorithm.

Our proposed algorithm abstracts from any particular application but simply considers work items to be assigned, consumed, split and donated. As in [1], it uses deques. Work items can be split into two (but any other number could have been taken) of which the sizes are not specified. The choice between finishing a work item and splitting it into smaller items is modelled as a non-deterministic choice to cater to a wider range of applications. We prove that this algorithm is deadlock-free and that termination happens if and only if all workers have finished their work load.

This algorithm was developed using the model checkers UPPAAL [13] and SPIN [14]. UPPAAL was mainly used to design the algorithm in a model-driven manner [15]. SPIN was employed for the verification of linear time properties and larger models. For small parameters (workload, number of workers), deadlock-freedom and termination if and only if all workers terminated, could be verified in UPPAAL and in SPIN. However, this model and a variant without non-progress cycles turned out to be too complex for exploring all the features of the algorithm. We therefore created a more abstract version by replacing the deque of a worker by the accumulated load, only. The resulting algorithm was explored in SPIN in depth and found to satisfy the required properties: no work gets lost or duplicated, deadlock-freedom, termination only in case of all workers completed their work, workers can donate or steal more than once, workers can take the role of a donor and a stealer in the same run, and different pairs of workers can engage concurrently in work stealing conversations.

This last, compact model is then enriched by the features of [2] to achieve fault tolerance. [2] uses ZooKeeper [16], an open source distributed coordination service, for detecting worker failures. ZooKeeper stores information in *znodes*. It allows setting *watches* on *znodes* that get triggered when the *znode* is deleted, or its data is changed, etc. All the workers create their respective znodes at the start of the computation and set a watch on the next worker's znode, thereby establishing a ring of watchers. When a worker fails, its znode disappears automatically and the watcher process gets notified by ZooKeeper. Thus, watches are used for monitoring workers in [2]. In the model, we added a ZKeeper process (for the ZooKeeper), a watch ring, the option of a worker to fail and a time-out if a donor failed to react. Znodes are not explicitly modelled, their behaviour is subsumed by the worker's.

For this extended model, we reconfirm the properties verified earlier where *termination* was adjusted to happen *if and only if all workers had either completed their work or had failed*. Additionally, we verified several other properties related to failure.

Up to our knowledge this is the first work stealing algorithm that has been modelled and verified with a model checker. The main challenge in modelling such algorithms is that the work load needs to be incorporated into the model and cannot completely be abstracted of. This leads to an explosion of the state space which grows in terms of the number of workers and the amount of work. However, to establish the most interesting properties of independent worker conversations four workers and four work items are sufficient. The details of the ideal algorithm, its pseudo code and the basic correctness proof is given in Section II. Section III introduces the various models and assumptions concerning the failure of a worker. Section IV summarizes the properties verified for the scenario of four workers and varying loads. It also contains the link to the SPIN models for self-exploration. The conclusions are given in Section V.

II. The Abstract Algorithm

The proposed algorithm for irregular tree processing assumes no failures of workers, the master or channels. We make the usual progress assumption: whenever a worker or master is at a location from which it can proceed, it eventually will do so. There are no shared data, and communications between master and workers and among workers themselves is solely based on message passing handshake communication. The master maintains an estimation of each worker's load (`estimate[worker]`) and two FIFO queues, `PR` and `PU` to store pending requests and pending updates, respectively. Each worker is equipped with a double ended queue `DQ` to buffer its current work. See the discussion in the introduction for the design decision to access the work item queue from both ends.

The system starts with the initial load allocation to the workers by the master. A load consists of several work items which are represented by its size. For example, the work load of a worker can consist of the work items 4,2,1,1 and its load would be 8, the sum of the sizes. Each worker keeps its work items in its double ended queue `DQ`. A work item may be either processed (`finishItem()`) or be split into two smaller items (`splitItem()`). The work

items for this internal work are taken from the bottom of DQ (popBot(DQ)). The splitted items are added again at the bottom of DQ (pushBot(item,DQ)). If a worker has processed all its items and its DQ is empty, it will request the master to send a donor id (requestDonor[worker]!). Based on its estimates, the master assigns a donor with the currently highest estimate or defers the worker's request by enqueuing it in the pending request queue PR or initialises termination of the overall system. Upon the receipt of a donor id, a worker will request this donor for work. If the donor has work to donate, it will take it from the top of its DQ (popTop(DQ)). However, it can also refuse to donate in case its workload is below a given threshold. In that case the work requester will ask the master to send another donor id. The master maintains its estimates by means of the updates given by the workers. Work loads below the threshold do not contribute to the estimate. However, the estimate of the work which, in principle, can be donated is never an underestimate. Workers update the master if the current workload has shrunk below half the load it had reported earlier to the master or if they have received a donation. Assigning a donor, the master immediately reduces its estimate of the donor's load by half. The master adds donor requesters to the pending updates queue PU to take record of which workers have an unknown work load.

It may happen that the master's overall estimation is zero though there is still work available. Such a situation can arise if a donation has been received but the receiver has not yet updated the master. If, before the update, the other workers finish all their work, the overall estimate is zero. A worker requesting a donor at this time will be added to the pending request queue. Therefore, after each incoming update, the master checks whether a worker in PR can be released by giving it some work.

The master initializes termination if its overall estimation is zero and there is no pending update (PU is empty). In the termination phase, a worker requesting for a donor will get the signal to terminate.

The pseudo code is given in Algorithm 1 and Algorithm 2. The functions and channels used are as follows:

Master: determineDonor() yields a worker with the highest estimate, zeroEstimate() is true if the accumulated estimate is 0, posEstimate() is true if the estimate is greater than 0, getFirst(PR) yields the first worker in PR without removing it, empty(PU) (empty(PR)) is true if PU = ε (PR = ε) and false otherwise,

Worker: pushBot(X,DQ) (pushTop(X,DQ)) adds X at the bottom of DQ (at the top of DQ), popBot(DQ) (popTop(DQ)) yields the bottom elenent of DQ (the top element of DQ) and removes it, finishItem() consumes a work item, splitItem(W) yields a pair of work items (W1,W2) where each of them is of half of the size of W,

Communication Channels:
between Master and Worker i: updateMaster[i](W), requestDonor[i], getDonor[i](D), terminateWorker[i], startWork[i] *between Workers i and j*: requestWork[i][j], getWork[i][j](W), refuseDonation[i][j]: the sending of a message via a channel is indicated with ! and the receipt with ?. If the message is not just a signal, the message contents is given by W for work items and by D for donors ids. The channel names are self-explanatory.

The pseudo code uses non-deterministic guarded selection statements as introduced by Dijkstra [17]. While it is easily understood that the algorithm helps to balance work, it is not obvious at all that the system is free of deadlocks. We prove deadlock-freedom and absence of premature termination in the next section.

Algorithm 1 Master's algorithm

```
1: PU:= ε, PR:= ε, threshold := C
2: terminatedMaster := false, terminationCount := 0, terminating := false
3: estimate[i] := W_i for all workers i
4: donor                                                  ▷ initially undefined
5: functions: zeroEstimate(), posEstimate(), empty(), getFirst(), determine-
   Donor()
6:
7: for all workers i do startWork[i]!(W_i)
8: while not terminatedMaster do
9:     if
10:        :: requestDonor[i]? =>
11:            estimate[i] := 0;
12:            remove i from PU;
13:            if zeroEstimate() and empty(PU) then
14:                terminating := true;
15:                terminateWorker[i]!
16:                terminationCount := terminationCount +1
17:            else
18:                if zeroEstimate() then
19:                    enqueue i in PR
20:                else
21:                    donor := determineDonor();
22:                    enqueue i in PU;
23:                    estimate[donor] := estimate[donor]/2;
24:                    getDonor[i]!(donor)
25:        :: updateMaster[i]?(W) =>
26:            remove i from PU;
27:            if W > threshold then
28:                estimate[i] := W;
29:                while not empty(PR) and posEstimate() do
30:                    j := getFirst(PR);
31:                    remove j from PR;
32:                    donor := determineDonor();
33:                    enqueue j in PU;
34:                    if estimate[donor]/2 > threshold then
35:                        estimate[donor] := estimate[donor]/2
36:                    else
37:                        estimate[donor] := 0;
38:                    getDonor[j]!(donor);
39:            else
40:                estimate[i] := 0;
41:                if zeroEstimate() and empty(PU) then
42:                    terminating := true;
43:        :: terminating =>
44:            while not empty(PR) do
45:                j := getFirst(PR);
46:                remove j from PR;
47:                terminateWorker[j]!;
48:                terminationCount := terminationCount + 1
49:        :: terminationCount = number of workers =>
50:            terminatedMaster := true
```

Algorithm 2 Worker i's algorithm

```
1: terminated := false
2: DQ := ε, pendWork := 0, lastUpdate:=0
3: functions: pushTop(), pushBot(), popTop(), popBot(), finishItem(), splitItem()
4:
5: wait for startWork?(W)
6: pushTop(W,DQ); pendWork := W, lastUpdate:=W
7: while not terminated do
8:     if
9:         :: not empty(DQ) =>
10:            if
11:                :: bottom(DQ) > threshold =>  W := popBot(DQ);
12:                   (W1,W2) := splitItem(W);
13:                   pushBot(W1,DQ); pushBot(W2,DQ);
14:                :: true =>  W := popBot(DQ); finishItem(W);
15:                   pendWork := pendWork - W
16:                :: pendWork < lastUpdate/2 => updateMaster[i]!(pendWork)
17:                   lastUpdate := pendWork
18:        :: empty(DQ) =>
19:            requestDonor[i]!
20:            while not terminated & donor not received do
21:                if
22:                    :: requestWork[j][i]? =>
23:                       refuseDonation[j][i]!
24:                    :: terminateWorker[i]? =>
25:                       terminated := true;
26:                    :: getDonor[i]?(donor) =>
27:                       requestWork[i][donor]!
28:                       if ...wait for donor response...
29:                           :: getWork[i][donor]?(W) =>
30:                              pushTop(W,DQ); pendWork := W;
31:                              lastUpdate := pendWork;
32:                           :: refuseDonation[i][donor]? =>  skip;
33:                       updateMaster[i]!(pendWork)
34:        :: requestWork[j][i]? =>
35:            if
36:                :: pendWork < threshold =>
37:                   refuseDonation[j][i]!;
38:                   updateMaster[i]!(pendWork)
39:                :: pendWork ≥ threshold =>
40:                   W := popTop(DQ);
41:                   getWork[j][i]!(W);
42:                   pendWork := pendWork - W;
43:                   updateMaster[i]!(pendWork)
44:                   lastUpdate := pendWork
```

Correctness and Deadlock Freedom

The FIFO queues of pending updates and of pending requests, PU and PR can be manipulated by the master only, and this can happen only in connection with the communication with a worker. In the following we prove deadlock freedom where the number of workers is assumed arbitrary but fixed. Note that live-locks are possible as will be discussed later.

Lemma II.1. 1) *If a worker is in PR then it is waiting for a donor or for the request to terminate (at line 21 or 23, worker alg.), and its work estimate maintained by the master is 0.*
2) *A worker is added to PR only after having been removed from PU (lines 12 and 19 of master alg.)*
3) *A worker is removed from PR only if it is assigned a donor (line 29 - 38, master alg.) or it is requested to terminate (line 47, master alg.). In the former case it is added to PU after its removal from PR.*
4) *A worker can request for a donor only if it is neither in PR nor in PU.*
5) *A worker is added to PU if and only if it gets a donor assigned (line 24 and line 37, master alg.). If this happens immediately on its request, the worker was not in PR by (4). Otherwise, the worker is removed from PR prior to its addition to PU (line 31, master alg.).*
6) *A worker can be in PU or in PR but not in both of them at the same time.*
7) *A worker is removed from PU only if this worker updates the master or it requests a donor.*
8) *If a worker is at line 8 (worker alg.) it cannot be in PU.*

Proof. 1) When a worker requests for a donor, it is at line 19 in the worker alg. It is waiting for a response from the master, which can be either donor id or request to terminate (line 20, worker alg.). The master adds the worker to PR only if it requests a donor (lines 10-19, master alg.). At the time of adding the worker to PR, the overall estimate is 0 (line 18, master alg.). This estimate is updated only when the worker sends an update. As the worker is waiting, the estimate continues to be 0 as long as the worker is in PR. A worker is removed from PR either before it gets a donor assigned or the request to terminate (line 31 and 42).
2) obvious.
3) If a worker is in PR by (1) it is at line 21 or 23 (worker alg.). The while loop at line 20 can be left only via going through line 25 or line 29. In the former case, the terminate message implies removal from PR by the master. In the latter case, the master had removed the worker from PR with the donor assignment.
4) This invariant can be proven by induction of the number of repetitions of the while loop at line 7 (worker alg.). Initially, PU and PR are empty. The worker is added to PU or PR only when it makes a donor request (lines 10, 19 and 22 in master algorithm). Thus, for the first time when the worker requests for a donor the property holds.
   After making a request for a donor, the worker enters the while loop at line 20. For the outer while loop to execute again (for the worker to make a request for donor), the worker must have a **getDonor[i]** event and then send an update to the master at line 33. This ensures that the worker is removed from PU. If the communication for the **getDonor[i]** event occurred via line 34 of the master algorithm then the worker had been removed from PR at line 31. If the communication was via line 24 then the worker had not been added to PR since it last entered the while loop body. Hence by induction hypothesis, it was not in PR.
5) by (2) and (4).
6) by (2), (4) and (5).

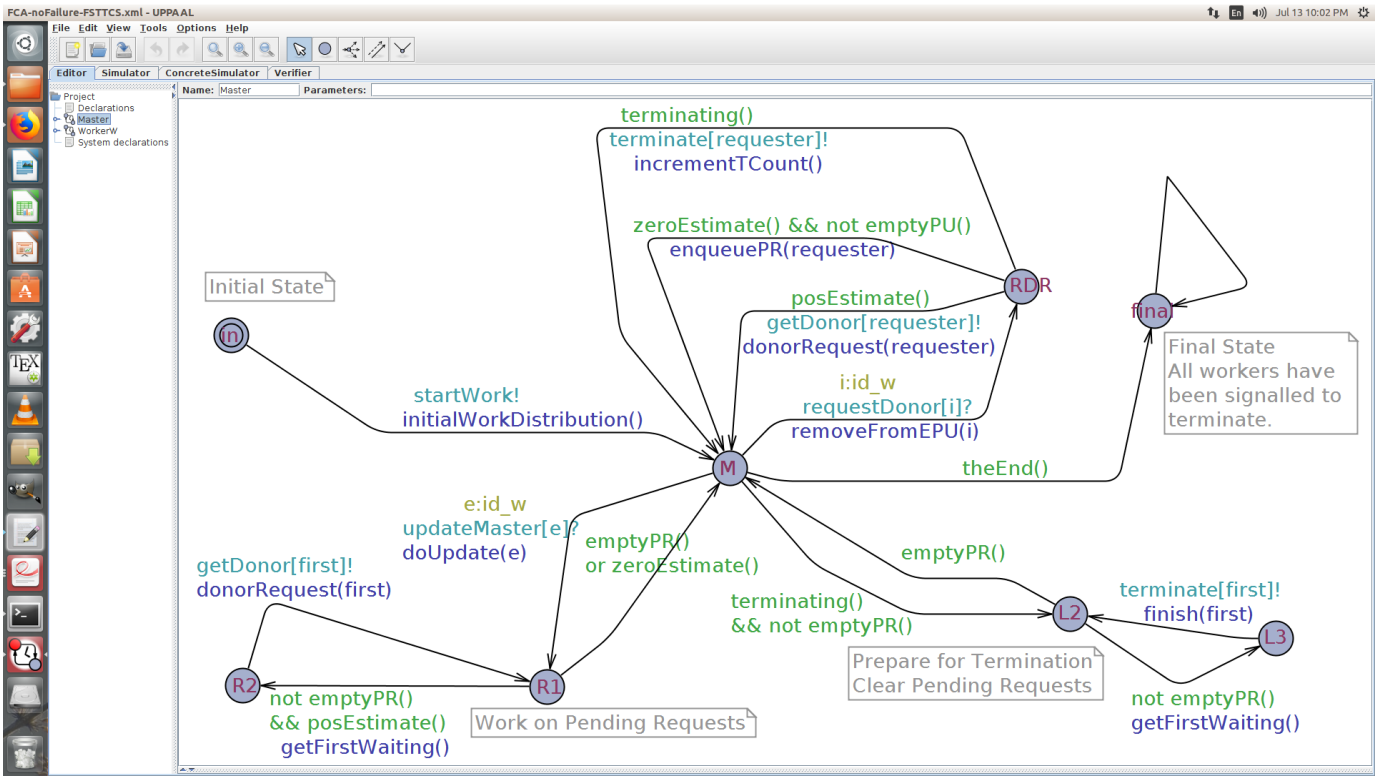

Fig. 1. The UPPAAL master template of **ITP** .

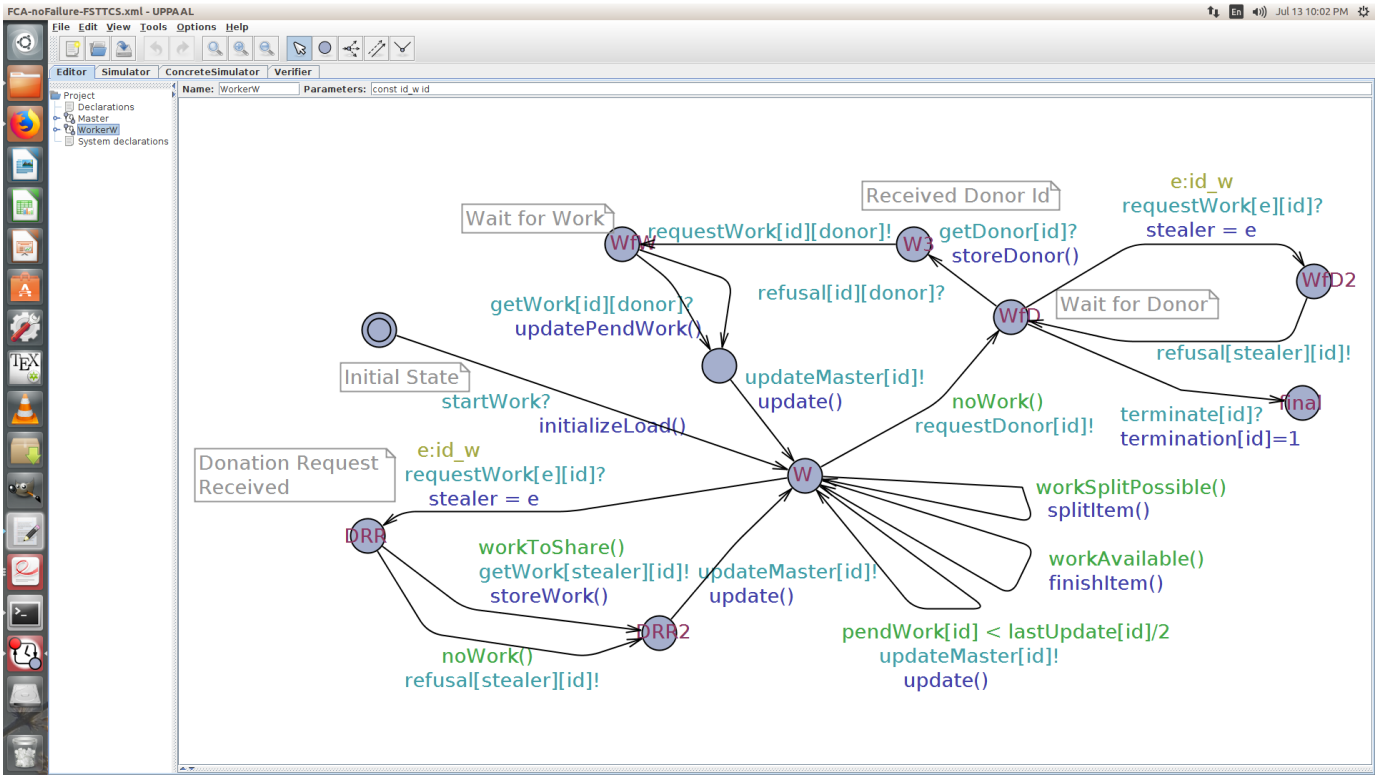

Fig. 2. The UPPAAL worker template of **ITP** .

7) by code inspection.
8) similar as the proof of (4). A worker is added to PU only when a donor gets assigned (5). When the worker receives the donor id from the master it is at line 26. It updates the master at line 33, before re-entering the body.

□

Lemma II.2. 1) *If two workers engage in a work donation then they will also complete this communication, that is, if a* `requestWork` *communication happens then there will also be an answer by the donor.*
2) *As long as the master has not terminated it can always complete the body of the selected option (i.e. it can return to line 9, master alg.).*
3) *If a worker is deadlocked it is either in PR (at line 21, worker alg.) or waiting to communicate with its donor (line 27, worker alg.) in its code.*

Proof. 1) follows directly from the code: there are no alternatives for a worker not to complete an interaction, and one of its guards always evaluates to true.
2) If there was a donor request (but no donor assignment, yet), the respective worker is at line 21 or 23 in its code and hence, able to receive the master's reply. Similarly, if there was an update event and a communication with a worker is required, then this is due to releasing a worker from PR. This worker at line 21 or 23 by Lemma II.1(4)). By (1) it is able to communicate.
3) A process can only deadlock if its communication partner is not responding. Thus, a worker can always finish its work items. By (1) and (2) it cannot be stuck between line 36-44. So if there is a deadlock, it must occur between lines 22-32. Again by (1) and (2), this can only happen at line 21 and line 27.

□

Theorem II.3. 1) *The algorithm is deadlock-free.*
2) *If the master has terminated all the workers have terminated and all work items have been consumed (up to the final threshold).*
3) *If all workers have terminated, the master will terminate as well.*

Proof. of 1. Suppose there is a deadlock. Let us first assume the overall state is terminating (`terminating = true` in Algorithm 1 line 42–43). The overall estimate must be 0 and PU must be empty. Further, no worker can be in PR (the master would have terminated all those workers by Lemma II.1(1) , Lemma II.2(2)). Hence by Lemma II.2(3), any deadlocked worker has to be waiting for a donor at line 27. However, by Lemma II.1(7), such a worker must be in PU, a contradiction. Hence, the overall state cannot be terminating.
So if there is a deadlock, the system is not terminating. Since workers cannot have terminated they must be stuck either in PR or be waiting for a donor at line 27 (Lemma II.2(3)). Hence, the overall estimate is 0 and PU is non-empty (otherwise the system would be terminating). Suppose there are k workers in PU. Out of these k workers, let i be the worker that joined PU last and let j be the worker that had been assigned as its donor. Since, all workers in PU have an empty estimate, and j must have had a positive estimate at the time of donor assignment, j could not have been in PU or PR. Hence, when j later requested a donor – as it couldn't terminate – the overall estimate was 0 (otherwise, j would have been added to PU). So it was added to PR and entered the while loop at line 20. But then it could have interacted with i via **requestWork[i][j]** and refuse to donate. So i was able to update the master after this and was removed from PU, contradicting that it was in PU at the time of the deadlock. □

Note that the algorithm allows for a work item being stolen and immediately donated repeatedly among a number of workers. However, this can be easily fixed by enforcing a worker to immediately work on an item stolen (by inserting the code between line 11 -17 at line 30 after line 30). Another solution is discussed in Section III.

III. The Models

The main objective of our work was to show fault tolerance of the system in [2] and, in broader terms, of our work stealing algorithm for irregular tree processing. We first modelled the system without any provision addressing worker failure. Though we had a mathematical proof of deadlock-freedom, it had to be ensured that the model was faithful to the pseudo code. We created three models for the fault-free case, increasing the level of abstraction to reduce the complexity of the system.

A. Models for the Ideal System without Worker Failure.

Our first model, **ITP** faithfully modelled the double ended queues and their manipulations (on an abstract level, barriers e.g. were ignored) with respect to the pseudo code. This model turned out to be too complex to verify the most interesting properties. None of our model checkers (we used SPIN and UPPAAL) could deal with four workers and a non-trivial load (at least four work items). Four workers are required to let two pairs of workers independently engage in a work donation conversation. By this we mean that if two workers have the load of two work items each and the other two none, the master can assign the former as donors to different workers. The two resulting work request/donate conversations are independent of each other. Though this could not be verified, still, the model was useful for simulations and for testing particular runs of the system. For three workers, the maximal workload that the model checkers could deal with was three work items.

In the sequel, whenever we refer to a particular model it is understood to be the SPIN model if not otherwise clarified. Work assignments are given as vectors $(v_1, \ldots, v_n)$ where n is the number of workers and v_i the (accumulated) initial work assigned by the master to worker i.

It was observed that the complexity of **ITP** was partially due to non-progress cycles. These are subruns which lead to configurations already gone through. A worker steals work and immediately donates it to the next worker who, again, donates

```
c1 [](initialisationDone -> noWorkLost)
c2 [](masterTerminated -> terminationNotified[0] && terminationNotified[1] &&
    terminationNotified[2] && terminationNotified[3] )
c3 <> masterTerminated
c4 <> (terminated[0] && terminated[1] && terminated[2] && terminated[3])
c5 [](stealerRoleCount[0]<2)
c6 [] !(stealerRoleCount[0]==1 && donorRoleCount[0]==1)
c7 [] !(stealerRoleCount[0]==1 && stealerRoleCount[1]==1
    && donorRoleCount[0]==0 &&  donorRoleCount[1]==0 && donorRoleCount[2]==0
    && donorRoleCount[3]==0 && donorOf[0]==3 && donorOf[1]==2))
cf4 <> ((terminated[0]||failure[0]) && (terminated[1]||failure[1]) &&
    (terminated[2]||failure[2]) && (terminated[3]||failure[3]))
f1 [] (!failure[3])   //to be refuted
f2 [] failureCount<N
f3 [] (failure[3] -> failureCount>=1)
```

Fig. 3. Claims to be verified or refuted.

it to the next worker. If this repeats until the particular work item is back at the first worker donating it, this constitutes a non-progress cycle.

Such non-progress cycles were allowed by the pseudo code as we did not want to constrain the implementation to when and where to enforce the progress (of work consumption or split) and to keep the code simple. In the implementation in [1] progress is ensured by enforcing a worker to immediately do some work when it receives a donation. For the pseudo code this means line 30 of the worker code needs to be replaced by a copy of lines 11-17.

In **ITP-nnPC** such non-progress cycles were removed by two changes:
(a) A worker is not permitted to donate work if there is only one work item in its deque (independently of the size of the item).
(b) If a worker refuses to donate a work item (because of (a)), it has to split or finish the item immediately after the refusal. The two constraints ensure that with each assignment of a worker as a donor, its accumulated work either reduces or the length of its deque increases by one. Hence, at its next donor assignment it either has less work or it cannot refuse to donate and by that it will reduce its load. Note, that (a) alone would would still permit non-progress cycles as the same donor can be assigned to the potential stealer after it had received a refusal earlier. Other non-progress cycles resulted from the way termination was communicated in **ITP** . These cycles were also removed by the master terminating all workers once there is no estimate and no pending update. The model was checked for absence of non-progress cycles.

With **ITP-nnPC** we were able to verify deadlock-freedom for four workers and the load of (0,0,2,2) and (0,0,1,3) while for load (0,0,0,4) the entire state space could not be explored by our SPIN system. However, it should be noted that the latter load assignment would yield symmetric load configurations (0,2,0,2) and (2,0,0,2), and (0,0,1,3),(0,1,0,3), and (1,0,0,3) for which the results were available.

To sum up, **ITP-nnPC** allowed us to verify deadlock freedom and independent engagements of 2 worker pairs. However, it was too complex to add any fault tolerant feature without going beyond the limits of our model checker system.

For the third model, **ITP-compact** we rigorously abstracted from the deques by simply maintaining the overall workload of a worker (master estimations were not affected by this). In case of a donation request, a worker either
(i) donates half of its pending work[1], or
(ii) reduces the pending work by 1 (processing work).
Work refusal would only happen in case of a work load less than or equal to 1. In the latter case the item has to be consumed immediately after the refusal to ensure progress.

As expected, **ITP-compact** significantly performed better than the other two. However, even for **ITP-compact** the maximal load for which we could verify deadlock-freedom for four workers was (0,0,2,4). Approximate time for verifying deadlock-freedom for four workers and the given load in the various models is given in Table I. We used SPIN on a DELL machine Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with 16GB RAM. We used exhaustive state space exploration with collapse compression, depth first search and partial order reduction. As directed by SPIN, extra compilation directives were applied when recommended which reduced the required memory on the cost of longer run-time. For all models it can be observed that there is a considerable increase of time and number of states when the load (0,1,1,2) is increased to (0,0,2,2). Several factors contribute to this. An individual load of 1 can only be consumed and not donated by a worker. It will not contribute to the estimate computed by the master. So with load (0,1,1,2), the master's initial estimate is 2, and with (0,0,2,2) it is 4. With load (0,0,2,2), two donations can happen independently. If a stealer delays updating the master, it may result in other workers being added to the pending request queue in varying order. All this contributes to the state space. Further, in the **ITP-nnPC** model, deques are modelled explicitly which increase the state vector. Compile directives given by SPIN lead to an immense slow down of the compilation. The arguments given also explain why the

[1]in case the load is odd, it is the floor of half of work. In this case it is assumed that one item has been processed before the donation.

increase of time and space from the loads (0,0,0,2) to (0,1,1,2) is small.

B. Model for the System with Worker Failure.

To verify the resilience property with respect to worker failure we incorporated into **ITP-compact** the possibility of a worker to fail, the watches of workers and a process known as ZKeeper to handle reports of failure (representing ZooKeeper). Again, to cope with the state space explosion, the resulting model is highly abstract.

The Failure Model The failing of a worker affects another worker only if it is engaged in a conversation with it. If a designated donor fails before it has delivered the work to the requesting stealer, the stealer needs to be released from the state of waiting and provided a new donor. A similar situation arises when the dedicated donor fails before the potential stealer can place its request. For both these situations time-outs are introduced which allow the worker to move on and request for a fresh donor as in the case of a donation refusal. In our model we therefore only incorporate the failing of a worker after it received a donation request. We do not model work recovery, in the model the work in the deque of a failing worker gets simply lost. Further, the master and ZKeeper are assumed not to fail at any point of a computation.

In the implementation of [2], a watch ring is established among the workers for detecting and recovering from worker failures (see Section I). When a worker fails, it is noticed by ZooKeeper by the absence of heart-beat. ZooKeeper then notifies the previous worker (alive) in the ring. That worker performs the recovery tasks to ensure at-least-once semantics of the system. ZooKeeper is a passive entity executing basic commands on the requests of workers, only, it is reliable and assumed not to fail. Hence, in our model, the two separate events are rendered as one, that is by the watcher observing the failure of a worker directly. After the repair work, the watcher requests a new watch from ZooKeeper which reacts with its delivery. These two events are not abstracted from as the watch ring is a relevant data structure for all processes and workers can fail concurrently. In **ITP-ZooKeeper** the book-keeping tasks (removing the failed worker from `pendPU` etc.) performed by the master in [2] (after having been notified by ZooKeeper) and the re-establishing of the watch ring, are done by ZKeeper. This can be seen as another way to speed up the progress of the overall computation or to reduce the state explosion. The delay the master would take in updating its records can be seen as modelled in the delay that ZKeeper takes to do it.

For potential stealers waiting for the assigned donor to communicate, time-out transitions have been introduced. After the time-out, the worker updates the master. This again is a form of progress enforcement as with the update the worker is considered as idle and termination can be announced in case there are no other pending updates.

IV. Verification Results

As the first model checker we had chosen UPPAAL. Its GUI helped considerably in the initial phase of the design. We verified the claims of Proposition II.3 for three workers and accumulated load 4. The UPPAAL model was then revised by removing non-progress cycles and adding the fault tolerant features. This allowed us to verify several properties for loads up to (0,0,2,2) for four workers. The individual load of 4 for a single worker, however, could not be handled.

As we later switched to SPIN where the main model checking was done, we are here only showing the automata models for a graphical visualisation of the algorithm 1 and 2. It should be easy to match the codes given in Algorithm 1 and 2 with these templates.

For the SPIN models, properties were verified for up to four workers and maximal loads of (0,1,1,2) in **ITP-nnPC** , (0,0,0,4) in **ITP-compact** , and (0,0,2,2) in **ITP-ZooKeeper** . Apart from deadlock-freedom and freedom of non-progress cycles, the following properties were established or refuted. The claims given in brackets refer to the LTL formulas in the PROMELA code (see Fig. 3).

1) No work is lost or added. (Claim[2] `c1`)
2) If the master terminates, then all the workers have been notified to terminate. (Claim `c2`)
3) Eventually the master terminates. (Claim `c3`)
4) Eventually all workers terminate. (Claim `c4`)
5) A worker can steal work more than once. (Claim `c5` to be refuted when the load is sufficient.)
6) A worker can act as a donor and as a stealer. (Claim `c6` to be refuted when the load is sufficient.)
7) Two pairs of workers can engage currently in a work donate conversation. (Claim `c7` to be refuted when the load is sufficient.)

Property 5 was checked for the first worker with the overall load (0,0,2,2). In **ITP-ZooKeeper** , the role of a worker as a stealer and as a donor in a single execution was established for the first worker with overall loads (2,2,0,0) and (0,0,0,4). With the latter it was verified that a worker can donate part of its stolen work. Property 7 was deduced by establishing that a state can be reached with workers 0 and 1 shown in the stealer role while the assigned donors have not yet acknowledged the donation by incrementing the donor role count. Both the role counts are increased by the donor.

For the fault tolerant model, apart from checking the absence of deadlocks, we re-verified the above properties of which the fourth had to be adjusted to the possibility of failure (claims `cf4`). It was then verified that up to three workers can fail within a run. The last worker can not fail as there is no watcher to communicate with in the model.

The core of the PROMELA code of **ITP-ZooKeeper** is shown in Figures 4, 5 and 6 in the appendix. Inline statements and the states `final` and `dead` are not shown. The variable

[2]The respective ghost variables and their manipulations are not shown in the code. `noWorkLost` states that the current overall pending work plus the number of work items consumed equals the initial overall work.

TABLE I

VERIFICATION RESULTS MODEL-WISE. THE FIRST NUMBER GIVES THE APPROXIMATE TIME FOR VERIFYING DEADLOCK FREEDOM FOR FOUR WORKERS AND THE GIVEN LOAD. THE SECOND ENTRY PROVIDES THE APPROXIMATE NUMBER OF STATES STORED. PARTIAL ORDER REDUCTION AND COLLAPSE COMPRESSION WAS APPLIED IN ALL CASES.

Model	(0,0,0,2)	(0,0,1,2)	(0,1,1,2)	(0,0,2,2)	(0,1,2,2)	(0,0,0,4)	(0,0,2,4)
ITP-nnPC	7.72 sec 3,825,000	8.52 sec 4,336,547	11 sec 4,937,513	12.5 min 2.703e+08	33 min 3.183e+08	$>$ 29 hours $>$ 1.0200e+09	? ?
ITP-compact	0.18 sec 110,835	0.19 sec 127,786	0.23 sec 148,475	3.26 sec 1,935,400	3.4 sec 2,041,920	4 min 99,889,400	$>$ 11 hours $>$ 3.630e+08
ITP-ZooKeeper	3.4 sec 2,202,000	3.7 sec 2,447,400	4.2 sec 2,666,300	50.7 sec 29,308,900	52.6 sec 30,249,700	33.4 hours 8.87e+08	? ?

`masterTerminated` is set by the master at the final state. All inline statements are atomic. To increase readabiliy, atomic enclosures in the core code which had been added to reduce the state space, only, are not shown. The claims are given in Figure 3 and the models can be accessed from our repository[3].

V. CONCLUSIONS

We have proposed and verified an abstract algorithm for distributed irregular tree processing aiming at efficiently dealing with the inherent computation skew. Our work can also be seen as a case study of model checking a distributed fault tolerant work stealing algorithm. The main issue was state explosion: a faithful modelling of the algorithm would not allow for the verification of essential properties. As the major remedy, we abstracted deques to size numbers. A similar bold abstraction step was taken in [18] where fault tolerant distributed algorithms with a crucial use of threshold guarded commands were to be model checked (e.g. Paxos). Compared to that work, we had to deal with three (not just one) kind of processes (master, ZKeeper and workers) and threshold guarded commands are nowhere used. Complexity-wise, the limiting parameters are the number of workers and the size and distribution of the work load.

Up to our knowledge this is the first work stealing algorithm that has been modelled and verified with a model checker. Formal verifications of deques can be found in [19], [20]. A simulator has been implemented in [21]. A verification of an entire algorithm we couldn't find. Though we have been able to verify characteristic properties, it is apparent that more complex algorithms and properties will be limited by the state explosion problem. The latter is inherent in work stealing algorithms as workers are designed to be exchangeable and this leads to many symmetric configurations. It is interesting to note that the irregular tree processing algorithm for concept mining from which we started out [1], is the result of breaking up symmetries in the computation of concepts. In particular, duplicate concepts are pruned. It seems that a similar strategy is applicable to break up the symmetries in the model checking. Further work would also explore a parameterized approach to the verification.

ACKNOWLEDGMENT

This research is supported by SPARC, a Govt. of India Initiative under grant no. SPARC/2018-2019/P682/SL.

[3]https://cloud.iitmandi.ac.in/f/a103fd904cb642f08ae7/?dl=1

REFERENCES

[1] S. Patel, U. Agarwal, and S. Kailasam, "A dynamic load balancing scheme for distributed formal concept analysis," in *Proc. of 24th IEEE Int'l Conf. on Parallel and Dist. Syst. (ICPADS'18)*. IEEE, 2018, pp. 489–496.

[2] A. Khaund, A. M. Sharma, A. Tiwari, S. Garg, and S. Kailasam, "RD-FCA: A resilient framework for distributed formal concept analysis," 2020, under submission.

[3] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1, pp. 5 – 33, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370201001084

[4] M. Dorigo, G. D. Caro, and L. M. Gambardella, "Ant algorithms for discrete optimization," *Artificial Life*, vol. 5, no. 2, pp. 137–172, 1999.

[5] G. Di Fatta and M. R. Berthold, "Decentralized load balancing for highly irregular search problems," in *Symposium on Computers and Communications*. IEEE, 2006.

[6] J. Yang and Q. He, "Scheduling parallel computationsby work stealing: A survey," *International Journal of Parallel Programming*, no. 46, pp. 173–197, 2018.

[7] U. A. Acar, G. Blelloch, and R. Blumofe, "The data locality of work stealing," *Theory of Computing Systems*, vol. 35, no. 3, pp. 321–347, 2002.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, oct 2016.

[10] F. Xie and A. Davenport, "Massively parallel constraint programming for supercomputers: Challenges and initial results," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 334–338.

[11] J. Jaffar, A. E. Santosa, R. H. C. Yap, and K. Q. Zhu, "Scalable distributed depth-first search with greedy work stealing," in *16th IEEE International Conference on Tools with Artificial Intelligence*, 2004, pp. 98–103.

[12] U. A. Acar, A. Charguéraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, p. 219–228.

[13] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL: Status and developments," in *CAV 97*, ser. LNCS, O. Grumberg, Ed., no. 1254. Springer–Verlag, 1997, pp. 456–459.

[14] G. J. Holtzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[15] M. Kühnrich, "Formal model-driven design of distributed algorithms," *Electronic Notes in Theoretical Computer Science*, no. 251, pp. 49–64, 2009.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9. Boston, MA, USA, 2010.

[17] E. W. Dijkstra, "Guarded commands, non-determinacy and formal derivation of programs," *Commun. ACM 18 (1975)*, vol. 18, no. 8, pp. 453–457, 1975.

[18] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder, "Towards modeling and model checking fault-tolerant distributed algorithms," in

Model Checking Software, E. Bartocci and C. R. Ramakrishnan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 209–226.
[19] S. Mutluergil and S. Tasiran, "A mechanized refinement proof of the chase–lev deque using a proof system," *Computing*, no. 101, 2019.
[20] C. van Kampen, "Formal automated verification of a work-stealing deque," January 2020. [Online]. Available: http://essay.utwente.nl/80687/
[21] M. Khatiri, D. Trystram, and F. Wagner, "Work stealing simulator," *CoRR*, vol. abs/1910.02803, 2019. [Online]. Available: http://arxiv.org/abs/1910.02803

APPENDIX

```
computeEstimation(cEst)
do
:: Q?requestDonor,id->
    estimate[id] = 0; detDonor(id); removeFromPU(id);
    computeEstimation(cEst)
    if
    :: (cEst <= threshold && lenPU == 0) -> terminating = true;
        T[id]!terminate; terminationNotified[id]=1;
        terminationCount = terminationCount + 1;
    ::  (cEst <=threshold && lenPU != 0) ->   // add to PR
       pendReq[lenPR]= id; lenPR++
    :: (cEst > threshold) ->
        donor = donorOf[id]; pendUpdates[lenPU] = id; lenPU++;
        estimate[donor] = estimate[donor]/2
        G[id]!getDonor;
        fi
:: U?updateMaster,id -> removeFromPU(id)
    if
    :: (pendWork[id] > threshold) -> estimate[id]=pendWork[id];
        computeEstimation(cEst);
R1:     if
        :: (lenPR !=0 && cEst != 0 ) -> //serve pending requests
            first = pendReq[0]; removeFromPR(first); detDonor(first);
            estimate[donorOf[first]] = estimate[donorOf[first]]/2;
            pendUpdates[lenPU]= first; lenPU++;
            G[first]!getDonor; goto R1;
        :: else -> skip
        fi;
    :: else -> estimate[id] = 0
        if
        :: cEst== 0 && lenPU == 0 -> terminating = 1
        :: else -> skip
        fi
    fi;
:: terminating  && terminationCount != N - failureCount ->
R2: if
    :: lenPR !=0 -> first = pendReq[0]; removeFromPR(first);
        T[first]!terminate;
        terminationNotified[first] =1; terminationCount = terminationCount + 1;
        goto R2;
    :: lenPR ==0 && terminationCount != N - failureCount ->
        i=0;
        do
        :: i< N && terminationNotified[i]==0 ->
            T[i]!terminate;
            terminationNotified[i]=1; terminationCount = terminationCount + 1; i++;
        :: i <N && terminationNotified[i]==1 -> i++;
        :: i ==N -> goto final
        od
    :: lenPR==0 && terminationCount==N - failureCount -> goto final
    fi
::  terminationCount == N - failurCount -> goto final
od
```

Fig. 4. The PROMELA code of **master** after initialisation in **ITP-ZooKeeper**

```
S?startWork;
do
:: (pendWork[_pid] > 0 ) ->  pendWork[_pid]-- // there is work
:: (pendWork[_pid] == 0 ) ->                 // there is no work
      if
      ::  T[_pid]?terminate ->  goto final;
      ::  Q!requestDonor,_pid
      fi
      do
      ::  W[_pid]?requestWork,workreply,stealer -> workreply!refusal
      ::  T[_pid]?terminate ->  goto final;
      ::  G[_pid]?getDonor ->
            if
            :: T[_pid]?terminate ->  goto final;
            :: W[donorOf[_pid]]!requestWork,workchannel,_pid;
               if
               :: failure[donorOf[_pid]] -> skip;   // donor not responding
               :: workchannel?response;  //treats both case: refusal, donation
                  if
                  :: T[_pid]?terminate ->  goto final;
                  ::  U!updateMaster,_pid; lastUpdate[_pid]=pendWork[_pid];
                  fi
               fi
            :: failure[donorOf[_pid]] -> U!updateMaster,_pid;
            fi
            break
      od
:: W[_pid]?requestWork,workreply,stealer ->
    if
    :: pendWork[_pid]==0 -> workreply!refusal;
        lastUpdate[_pid]=0;
        if
        :: T[_pid]?terminate -> goto final;
        :: U!updateMaster,_pid;
        fi
    :: pendWork[_pid]==1 -> workreply!refusal;
        pendWork[_pid] = 0; lastUpdate[_pid]=0;
        if
        :: T[_pid]?terminate -> goto final;
        :: U!updateMaster,_pid;
        fi
    ::  pendWork[_pid]>1 -> add= pendWork[_pid]/2;
        pendWork[_pid]=add; pendWork[stealer]=add;
        stealerRoleCount[stealer]++; donorRoleCount[_pid]++; //for verification only
        workreply!getWork; lastUpdate[_pid]=pendWork[_pid];
        if
        :: T[_pid]?terminate -> goto final;
        :: U!updateMaster,_pid
        fi
    :: FF[_pid]!failed -> goto dead;
    fi;
:: FF[currentWatch[_pid]]?failed ->
    failer = currentWatch[_pid]; NF!_pid,failer;
    GW[_pid]?newWatch; currentWatch[_pid] = newWatch;
od
```

Fig. 5. The PROMELA code of **worker** in **ITP-ZooKeeper** .

```
do
:: NF?reporter,failer -> estimate[failer]=0;
    detWatch(failer); // provides failerWatch
    removeFromWR(failer); removeFromPU(failer); removeFromPR(failer);
    terminationNotified[failer]=1; failureCount++; failure[failer] = 1;
    currentWatch[reporter]=failerWatch; GW[reporter]!failerWatch;
:: terminating -> break;
od
```

Fig. 6. The PROMELA code of **ZKeeper** in **ITP-ZooKeeper** .