

Formal Methods Report  
A comparison between TLA<sup>+</sup>/TLC and Promela/SPIN

José Faria

June 2005

## **Abstract**

To compare different formalisms may be a complex task. In this report we present an evaluation done through the use of a case study: the algorithm presented by Harris [4] to deal with non-blocking linked-lists. The formalisms and corresponding tools studied were TLA<sup>+</sup>/TLC [11, 12] and Promela/Spin [6].

The presentation is globally self-contained — both the algorithm of the case study and the languages TLA<sup>+</sup> and Promela are explained with reasonable detail. No significant assumptions of previous knowledge are made.

# Contents

<b>1</b>	<b>Case Study</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.2	The solution presented . . . . .	3
1.3	Remarks . . . . .	5
1.4	Drawbacks . . . . .	5
1.5	Related Work . . . . .	6
<b>2</b>	<b>TLA<sup>+</sup> and TLC</b>	<b>7</b>
2.1	The Language / Tool (Short presentation) . . . . .	7
2.2	Going in more detail - Overview of TLA <sup>+</sup> . . . . .	8
2.3	Model developed . . . . .	11
<b>3</b>	<b>Promela and SPIN</b>	<b>17</b>
3.1	The Language / Tool . . . . .	17
3.2	Going in more detail - Overview of Promela . . . . .	17
3.3	Model developed . . . . .	20
<b>4</b>	<b>Comparison</b>	<b>25</b>
<b>A</b>	<b>TLA<sup>+</sup> model of Harris' Algorithm</b>	<b>31</b>
<b>B</b>	<b>Promela model of Harris' Algorithm</b>	<b>37</b>

# 1 Case Study

## *A Pragmatic Implementation of Non-Blocking Linked-Lists*

To make this document self-contained we present here the original algorithm used as the case study. For the original description please refer to [4]. This description can also be considered complementary to the one in the cited reference, once the textual explanations are rather short there: the algorithms are presented through pseudo-code.

### 1.1 Introduction

Linked-lists are a basic structure used in program design. A singly linked list can be simply defined as a chain of structures which contain a pointer to the next element. Each element is commonly referred to as a node. We will consider a node as containing two fields: a *key* field, as the identifier of the node, and a *next* field containing a reference (pointer) to the next node in the list.

In any ordinary distributed system we could have the scenario where different, independent processors would try to operate a common list. The two major operations that a processor can do are (1) inserting and (2) deleting a node to/from the list.

Both operations comprise locating the place where to insert/delete the node to/from the list and then *physically* perform the operation. A node is inserted by, after identifying its predecessor and successor nodes, making it point to the successor and changing the predecessor's reference to point to the new node. Deleting a node can simply be achieved by making its predecessor point to its successor, eliminating the references to itself.

Several problems with concurrent access to the list can arise when, e.g., (a) different processes try to insert nodes with the same identifier, (b) different processes try to delete nodes with the same identifier, (c) different processes try to insert nodes with different identifiers but the same predecessor and successor, (d/e) a process tries to insert a node whose identified successor/predecessor is being deleted by another process.

Traditionally, these and similar problems are solved by granting exclusive access to the shared resource (the linked list in this case), which stops/blocks all other processes trying to access to the same resource to evolve. By contrast, an implementation is non-blocking (or lock-free) if some process must complete an operation after the system as a whole takes a finite number of steps, which means that some process will always make progress despite arbitrary halting failures or delays by other processes [5].

The algorithm presented by Harris [4], which we will analyze in the following of this document, is able to deal with linked-list in a non-blocking fashion.

## 1.2 The solution presented

To manage the (singly linked) list in an easier way a usual simplification practice was followed: two special root nodes, "*Head*" and "*Tail*" were introduced. *Head* is always kept as the first element of the list and *Tail* as the last one.

The algorithm makes use of the atomic primitive compare-and-swap (CAS)<sup>1</sup>. CAS (*addr*, *old*, *new*) atomically compares the contents of a location *addr* with an expected value *old* and, if they match, writes the value *new* to that location. CAS returns a boolean indicating whether the substitution took place or not.

With the CAS operation the problems identified above as (a) and (c) can be solved straightforwardly. The *physical* insertion/deletion of a node is only achieved when the reference of its predecessor is changed. This is done atomically with CAS. If, at the moment when the instruction is executed, some other process made a *significant* change to the list, CAS will fail, and the process restarts by relocating again its predecessor and successor nodes.

Looking into the inserting process in more detail:

- (Ii) The node to be inserted *current* is generated;
- (IIi) The predecessor *left* and successor *right* nodes are identified (all the nodes have a key and they are stored in the list in ascendant order). The addresses of these nodes are stored in two variables. The *left* node is the node that has the biggest key strictly smaller than the *current* node key. The *right* node is the node that has the smallest key greater or equal than the *current* node key.
- (IIIi) The uniqueness of the node to insert is checked: if the key of the *right* node is equal to the one of the *current* node, the process is aborted; otherwise it continues its execution.
- (IVi) The *current* node is made to point to the *right* node (by a simple assignment instruction)
- (Vi) The *physical insertion* of the node, recurring to CAS (*addr*, *old*, *new*), is attempted: the node that is being presently pointed at that instant by the *left* node [*addr*] is compared to the one previously identified [*old*], whose value has been stored in a variable and:
  - If they match, the *left* node is made to point to the *current* node, concluding the inserting process.
  - If they don't match (which means that some change in the list has been made in the mean time), the process goes back to (IIi).

---

<sup>1</sup>See section 1.5 for more details about the support of this operation by existing processors.

With the procedure just described we could overcome the problems identified above as (a) and (c). However, the execution of deletions needs extra careful. For that, a new mechanism, omitted until now, will be introduced. Let's access its necessity with an example: two concurrent processes are, respectively, trying to insert and delete a node. These nodes are such that the node to be inserted has as predecessor the node to be deleted. If the insertion process terminates first but still late enough that the deleting process *doesn't see it* when it was identifying its successor and predecessor nodes, when the node is deleted<sup>2</sup> the new one that had just been inserted will be lost, causing a violation of the intended result.

To solve this, Harris' algorithm separates the deletion of a node in two phases, using a CAS for each one. The first one is used to *mark*<sup>3</sup> the node and the second one to *excise* it from the list. The node is considered *logically deleted* after the first stage and *physically deleted* after the second.

A marked node is still part of the chain of the linked list, but its marking is used to signal concurrent processes to not introduce new nodes immediately after those that are logically deleted.

Like this, in both operations (insert and delete), the left and the right nodes are actually not determined as said before (IIi) but instead as being: the left node the unmarked one that has the biggest key strictly smaller than the current node's key; the right node the unmarked one that has the smallest key greater or equal than the current node's key. This rule overrides the one mentioned in (IIi).

Description of the deleting process:

- (Id) an identifier key *value* is selected;
- (IIId) the list is searched in order to identify the *left* and *right* nodes. The rule is as mentioned just above: the *left* is the unmarked node that has the biggest key strictly smaller than *value*; the *right* is the unmarked node that has the smallest key greater or equal than *value*.
- (IIIId) the identity of the right node is checked: if its key is equal to *value* the process continuous to execute and the right node is the node to be deleted; otherwise it aborts<sup>4</sup>.
- (IVd) a new *neighborhood node* is identified: the immediate successor of the right node is stored in a variable *right\_next*.

---

<sup>2</sup>Recall that a node is deleted by simply making its predecessor point to its successor, eliminating the reference to itself.

<sup>3</sup>The actual implementation of this marking can be done recurring to an unless unused bit of the field next of the node.

<sup>4</sup>This mail happen for two reasons: (1) another concurrent process could have deleted the node in the mean time; or (2) there was some error in the assignment of the node to delete and it simply doesn't belong to the linked list.

**(Vd)** the *marking* of the (right) node is attempted (atomically, with the CAS operation): if the right node next field  $[addr]$  is still pointing to the node previously identified *right\_next*  $[old]$ , then its value is rewritten as marked  $[new]$ . If not the process goes back to (IId). This operation is guarded by the pre-condition that *right\_next* is not a marked node; otherwise the process goes directly back to (IId).

**(VIId)** the *physical deletion* of the (right) node is attempted (again atomically, with CAS): if the left node next field  $[addr]$  is still pointing to the right node  $[old]$ , then it is made to point to the node identified as its immediate successor *right\_next*  $[new]$ . Once again if this fails the process goes back to (IId).

### 1.3 Remarks

The textual presentation of the algorithm made here was, in general, quite complete and compliant with the original one [4], presented in pseudo-code. However, a significant simplification was made when reporting the identification of the left and right nodes (*IIi*) and (*IIId*). This was made because the same simplification was applied when constructing the models in TLA<sup>+</sup> and Promela. In the original algorithm this process is more elaborated and it is executed recurring to a *search* routine that incorporates a CAS operation to remove marked nodes between the left and right nodes, before returning.

### 1.4 Drawbacks

In Harris' algorithm a node, after being deleted from the list, may still be accessed. In fact, in some scenarios, this may be necessary, which stops the algorithm from using simple and efficient lock-free memory management methods [14]. Let's illustrate it with an example, previously identified as "problem (b): different processes try to delete nodes with the same identifier". We'll limit this illustration to two processes.

#### With memory release

- (1) Process 1 is assigned to delete a node with key  $k$  (*Id*), that belongs to the linked list.
- (2) Process 1 identifies the neighborhood of the node  $n$  (*IIId*, *IIIId*, *IVId*).
- (3) Process 2 is assigned to delete a node with the same key  $k$ .
- (4) Process 2 identifies the same neighborhood of the node  $n$  (*IIId*, *IIIId*, *IVId*).
- (5) Process 1 marks the node  $n$  (*Vd*).

- (6) Process 1 excises the node  $n$  (*VId*).
- (7) Process 2 tries to mark the node  $n$  (*Vd*) but the memory location of the node is now empty<sup>5</sup>, causing an error<sup>6</sup>.

**Without (or "with sufficiently delayed") memory release**

- (1) to (6) Same as before.
- (7) Process 2 tries to mark node  $n$  (*Vd*), but the comparison of the CAS operation fails<sup>7</sup>, and the process is sent back to (*IId*).
- (8) Process 2 identifies the new left and right nodes (*IId*).
- (9) Two different things may happen now: (1) *normally* the key of the new right node will be bigger than  $k$  and the process will end its execution (*IIId*). (2) If, in the mean time, another process has inserted back a node with that same key  $k$ , process 2 will detected it and eventually delete it.

## 1.5 Related Work

Due to the natural advantages of lock-free implementations over the traditional use of locks, this is been a subject of increasing interest. The atomic primitive CAS plays a central role in this subject. Like this, it is now supported by several processors, e.g. Sparc [20] and Intel [8]<sup>8</sup>. Another primitive commonly used is LL/SC/VL, *Load-Link/Store-Conditional/Validate* (see, for example, [9, 13, 17]).

In [14] it is pointed out Harris's algorithm problem in memory management<sup>9</sup> as a serious drawback to it. The same author presents, in [15], what he claims as "the reclamation memory" method.

A problem that affects many lock-free algorithm is the called "ABA problem" [7]. The atomic primitives LL/SC/VL seem to be a more convenient

---

<sup>5</sup>"Empty" is not an exact term. Once the memory location has been made free it can be in a multiplicity of different states that may lead to the referred error.

<sup>6</sup>The actual behavior of a process in this scenario depends on the details of the implementation.

<sup>7</sup>Recall that the comparison is made between the value of the node's next field previously recorded and the present one. Even if the composition of the list hasn't changed in the mean time, the values won't match because the present reference is marked while the value previously recorded is not. This is due to decision of marking a node by a change in an unless unused bit of the next filed of the node. In any case, if other implementation scheme wouldn't cause this comparison to fail the process would continue to (*VId*) and would then forcedly be sent back to (*IId*), continuing as announced in (8).

<sup>8</sup>This is presented as the operations "CASA" and "CASXA" in the Sparc processor, and as "cmpxchg" in the Intel's.

<sup>9</sup>See section 1.4

way of developing ABA problem free algorithms [16]. [13] presents an algorithm immune to this problem using LL/SC/VL. [3] presents a solution with CAS.

[2], from 1978, is a novel introduction to the subject of writing algorithms with exclusion and synchronization constraints as low as possible. Strangely enough no reference to it was found in this *new generation* of algorithms.

## 2 TLA<sup>+</sup> and TLC

### 2.1 The Language / Tool (Short presentation)

TLA<sup>+</sup> is a specification language for describing and reasoning about asynchronous, nondeterministic, concurrent systems [11, 12]. TLC is an explicit-state model checker for specifications written in TLA<sup>+</sup> [12, 21].

A specification in TLA<sup>+</sup> is summarized in a single formula that describes a state machine in terms of an initial condition, a next state relation and, possibly, some liveness conditions, which writes:  $Init \wedge \Box Next \wedge Liveness$ , where, in more detail<sup>10</sup>:

**Init** is the initial-state predicate—formula describing all legal initial states.

**Next** is the next-state relation, which specifies all possible steps (pairs of successive states) in a behavior of the system. It is a disjunction of actions that describe the different system operations. An action is a mathematical formula in which unprimed variables refer to the first state of a step and primed variables refer to its second state.

**Liveness** is a temporal formula that specifies the liveness (progress) properties of the system as the conjunction of fairness conditions on actions.

TLC<sup>11</sup> can be used to check safety and liveness properties of a specification written in the form  $Init \wedge \Box Next \wedge Liveness$ . For safety properties, TLC explores all reachable states in the model, looking for one in which an invariant is not satisfied or deadlock occurs (there is no possible next state). When TLC detects an error, a state trace that exhibits the error (*counter-example*) is printed as part of the error report. For a violation of a safety property, the error trace is guaranteed to be minimal-length. Because TLA<sup>+</sup> is such a high-level language, TLC is expected to be slower than comparable model checkers.

---

<sup>10</sup>Description from [10]

<sup>11</sup>Still, description from [10]

## 2.2 Going in more detail - Overview of TLA<sup>+</sup>

Because TLA<sup>+</sup> doesn't resemble any programming language, a model may have a "strange look" at a first glance. Therefore, we present here a description of the basic components of a TLA<sup>+</sup> specification through an example (Figure 1).

The example used is a simple model that manages a list of elements (keys). The possible operations are: (1) insert and (2) delete elements from the list. The list is structured as a sequence. Insertion is done by appending the new element to the bottom of the sequence. Deletion consists in simply removing the first element from the top of the list.

```

1 |----- MODULE Intro -----|
2 | EXTENDS Naturals, Sequences |
3 | CONSTANT Keys, N | Keys and N are both defined in the config file |
4 | VARIABLES list |
5 | ASSUME ( $N \in \text{Nat}$ )  $\wedge$  ( $N > 0$ ) |
6 | TypeInvariant  $\triangleq$   $list \in \text{Seq}(\text{Keys})$  |
7 |-----|
9 | Init  $\triangleq$   $list = \langle \rangle$  | List initialized empty |
12 | Insert(key)  $\triangleq$   $\wedge$   $Len(list) < N$  |
13 |            $\wedge list' = \text{Append}(list, key)$  |
16 | Delete  $\triangleq$   $\wedge$   $Len(list) > 0$  |
17 |            $\wedge list' = \text{Tail}(list)$  |
21 | Next  $\triangleq$   $\vee \exists k \in \text{Keys} : \text{Insert}(k)$  |
22 |            $\vee \text{Delete}$  |
24 | Spec  $\triangleq$   $Init \wedge \square [Next]_{list}$  |
25 |-----|
26 | THEOREM Spec  $\implies$   $\square \text{TypeInvariant}$  |
27 |-----|

```

Figure 1: Example of a specification in TLA<sup>+</sup>

The specification starts with

```

|----- MODULE Intro -----|

```

that begins a module called *Intro*. TLA<sup>+</sup> specifications are partitioned into modules. We use only one in this example<sup>12</sup>.

Line 2 introduces **EXTENDS**, a command to establish relations between different modules. By extending the modules *Naturals* and *Sequences* we can use the operations defined into them. *Naturals* is a module that contains the definitions of the "usual operators on natural numbers", like '+' or '>'.  


---

<sup>12</sup>Plus the standard modules *Naturals* and *Extends* as we shall see just bellow.

The *Sequences* module defines operations on finite sequences. The ones used in this module were `Append` and `Tail`. `Append(s, e)` appends element `e` to the end of sequence `s`. `Tail(s)` is the tail of sequence `s`, i.e. `s` with its first element removed.

`CONSTANT` introduces the parameters *Keys* and *N*. *Keys* is a set used to define the elements that can be part of the list. *N* defines the maximum length of the list. When verifying the model with TLC we need to explicitly define these parameters. This is due to the fact that TLC (as SPIN) only handles systems with a finite number of states. The definition is done in a separate file – the *config file* (Figure 2).

`VARIABLES` *list* declares the single variable of the module, *list*. Note that TLA<sup>+</sup> is an untyped language! The variables are simply listed, not typed.

`ASSUME (N ∈ Nat) ∧ (N > 0)` asserts that we are assuming that *N* is a positive natural number.<sup>13</sup>

$$\text{TypeInvariant} \triangleq \text{list} \in \text{Seq}(\text{Keys}) \quad (\text{Line 6})$$

defines the correctness criteria of the model: at any state, the list has to be a valid sequence whose elements all belong to the set *Keys*.

The line



is purely cosmetic, it can appear anywhere.

```
(*****
(*                               CONFIG Intro.cfg                               *)
*****)
SPECIFICATION Spec
(* This statement tells TLC that Spec is the specification to be
checked. *)
INVARIANT TypeInvariant
(* This statement tells TLC to check that TypeInvariant is an
invariant of the specification. *)
(*****
(* TLC requires that every declared constant in the specification *)
(*be assigned a value by a CONSTANT statement in the configuration*)
(*file. *)
*****)
CONSTANT
N = 3
Keys = {10, 20, 30}
(***** End of config file *****)
```

Figure 2: Configuration file

Once explained the *header* of a TLA<sup>+</sup> module, a specification is better understood if we read the rest of it from the bottom to the top!

<sup>13</sup>This has no effect in the definitions made in the module; it can, however, be taken as an hypothesis in the construction of proofs.

THEOREM  $Spec \implies \Box TypeInvariant$  (Line 26)

is the statement that represents our ultimate goal: our program ( $Spec$ ) fulfills all the correctness claims that we want to be verified ( $TypeInvariant$ ).

Moving up:

$$Spec \triangleq Init \wedge \Box [Next]_{list}$$

is the specification of our program. It tell us that our systems is a state machine, whose initial state is  $Init$  and that can evolve through the steps defined in  $Next$ .<sup>14</sup> The  $[ ]_{list}$  surrounding  $Next$  is used to validate *stuttering steps*: steps that leave the values of the variables unchanged.<sup>15</sup>

We have now only left to define  $Init$  and  $Next$ .  $Next$  is defined as the disjunction ( $\vee$ ) of the actions  $Insert$  and  $Delete$ :

$$Next \triangleq \vee \exists k \in Keys : Insert(k) \vee Delete$$

$Insert$  has as parameter the value of the element ( $key$ ) to insert in the list. For an insertion to occur there has to be ( $\exists$ ) a  $key$ , belonging to the set  $Keys$  to insert. Hence the expression  $\exists k \in Keys$  preceding  $Insert(key)$ .

The definition of an action is the conjunction of its guarding conditions, eventual internal computation and the actualization of the variables of the system (where the new value is represented by priming (')). All the variables need to be listed. The ones that are not affected by the action are listed as UNCHANGED. Example:

$$\begin{aligned} Action(parameters) \triangleq & \wedge var1 = TRUE \quad \text{first guard} \\ & \wedge var2 > var4 \quad \text{second guard} \\ & \wedge var1' = FALSE \quad \text{first actualization} \\ & \wedge var3' = var2 + var3 \quad \text{second actualization} \\ & \wedge UNCHANGED \langle var2, var4, var5 \rangle \quad \text{unchanged variables} \end{aligned}$$

The definition of the action  $Insert(key)$  is as follows:

$$\begin{aligned} Insert(key) \triangleq & \wedge Len(list) < N \\ & \wedge list' = Append(list, key) \end{aligned}$$

It is guarded by the condition the list didn't reach its maximal length and its effect is to append the new  $key$  to the list.<sup>16</sup>

Delete is guarded by the condition that there is some element in the list (the length of the sequence is greater than zero) and removes its first

<sup>14</sup>Note that no Liveness conditions were specified in this module.

<sup>15</sup>In case there were more variables in the module, they should all be listed here, as a tuple with the variables separated by comas:  $[Next]_{\langle var1, var2, \dots, varn \rangle}$

<sup>16</sup> $Len(s)$  is a function also defined in the module *Sequences*. It returns the length of the sequence  $s$ .

element using the operation *Tail*, explained above:

$$\begin{aligned} Delete \triangleq & \quad \wedge Len(list) > 0 \\ & \quad \wedge list' = Tail(list) \end{aligned}$$

Finally, the list is initialized empty:

$$Init \triangleq list = \langle \rangle$$

For a nicer and easier to read presentation, all the comments in a TLA<sup>+</sup> module are shaded.

A module terminates with the symbol

## 2.3 Model developed

In this section we present the developed TLA<sup>+</sup> model of Harris' algorithm. The design decisions are explained and some extracts of the model are shown. The full model is in the Appendix A.<sup>17</sup>

The central aspect of writing a (TLA<sup>+</sup>) specification is choosing the level of abstraction. This implies choosing the variables that represent the system and choosing the granularity of the steps that allow the system to evolve. The granularity chosen is very simple to summarize: it basically follows the textual description made in section 1.2, where each of the phases there identified as Ii, Iii, ..., Vi; Id, ..., Vd, Vid corresponds to an atomic action in the TLA<sup>+</sup> model.<sup>18</sup>

Two variables were chosen: *mem* and *proc*. *mem* represents the memory model of the system: a series of memory addresses where nodes can be recorded. The list of addresses is enumerated in the constant *Adr* and the nodes are represented as a record of three fields: *key*, *next* and *mark*. The set of all possible keys is defined in *Keys*. The *next* field (when not empty) should contain an address – to represent the pointer of a node to its successor. *mark* is a boolean variable – a node can only be marked or not. *mem* is defined as a function of addresses to nodes.

*proc* is a variable that encapsulates all the information to deal with the behavior of the system. This information is, on its turn, encapsulated in four records: *ninfo*, *procIns*, *procDel* and *choice*. *ninfo* registers the information that effectively needs to transit in between intermediate steps, i.e. the key and next values of the node to insert/delete and its neighborhood.

<sup>17</sup>The description made here assumes the previous reading of section 2.2 or former familiarity with TLA<sup>+</sup>. The full model is enriched with many comments, so that, after some familiarity with TLA<sup>+</sup>, it can practicably be read with no further explanations.

<sup>18</sup>Strictly speaking, there's an exception: IId and IIId where grouped in a single action.

*procIns*, *procDel* and *choice* are "artificial variables" of the model, i.e. they are used to control aspects that are not directly connected to the algorithm, but to the TLA<sup>+</sup> model itself. Because the insertion and deletion processes were both modeled in several different actions, to ensure the correct sequence of execution an extra mechanism is needed. This was done recurring to a variable that registers in what intermediate step the execution is. *procIns* is that variable for the insertion process and *procDel* for the deletion one. *choice* identifies the commitment of a process (after being committed a process is only set free after finishing all the intermediate steps).

Because we want to have a system with multiple processes running, the variable *proc* is made a function of processes to its referred four records (*ninfo*, *procIns*, *procDel* and *choice*). The set of existing processes is defined in the constant *Process*. Figure 3 shows the header of the model. Figure 4 shows a possible instance of the configuration file.<sup>19</sup>

*TypeInvariant* contains the definition of all the variables. It is made an invariant of the system, which means that any valid step of the specification has to keep the defined structure invariant.

Once understood the variables abstraction, we should move to the bottom of the module (Figure 5). Its reading is straightforward. The theorem of the model is that our specification implies the invariants *TypeInvariant* and *Coherence*<sup>20</sup>. *Coherence* expresses the order relation that should be kept in between the nodes. It states that all the nodes registered in *mem* and that point to another node, have a *key* that is strictly smaller than the key of the node that they point to (Figure 6).

The specification *Spec* of the system is defined by the initial conditions expressed in *Init* and the next-state relation *Next*. *Next* is a disjunction of the actions *SetInitNodes*, *Insert* and *Delete*. *SetInitNodes* is the "artificial action" where the special root nodes *Head* and *Tail* are inserted into the list. *Insert* and *Delete* both take as parameters a *process* (responsible for the execution) and a *key* (identifying the node to insert/delete). Both actions are defined as a disjunction of five finer grained steps. Figure 7 shows the correspondence in between them and the textual description made in section 1.2.

As a representative example of all the other actions, we'll cover the definitions of *CreateI*, *LocateD* and *CasD2*.

*CreateI(p, key)* (Figure 8) takes as parameters a process *p*, to execute the action, and the *key* of the node that we want to insert. It's guarded

---

<sup>19</sup>Note the use of "possible instance". In fact the configuration file may have many instances by changing the definitions of the constants in it. The TLA<sup>+</sup> model remains, however, unchanged.

<sup>20</sup>More rigorously, it asserts that the formula follows logically from the definitions in this module, the definitions in the extended modules *Naturals*, *Sequences* and *FiniteSets*, and the rules of TLA<sup>+</sup>.

```

1 |----- MODULE HarrisR -----|
  |
  | This model represents the algorithm presented by Harris for the implementation of non-blocking linked-
  | lists.
  |
  | In this model the nodes inserted to the list are stored in the global variable “mem”. The actions of
  | deleting and inserting nodes to the list are divided, for a finer-grained solution in several intermediate
  | steps.
  |
  | For ‘artificial’ (to be seen in some comments below) we mean all the aspects that are not directly
  | connected to the algorithm, but that represent adaptations of it to be possible/more easily modeled.
10 | EXTENDS Naturals, Sequences, FiniteSets
11 | CONSTANT Adr,           the set of addresses
12 |           Keys,         the set of keys
13 |           Process,      the set of processes
14 |           HEAD, TAIL  values of the keys of ‘Head’ and ‘Tail’
  |
16 | VARIABLES mem,         ‘state of the memory’, with all the nodes inserted in the Linked List
17 |           proc,        auxiliary variable – information for the intermediate stages
18 |           setup       ‘artificial’ variable for the initial insertion of ‘Head’ and ‘Tail’
  |
20 | ASSUME  HEAD has to be smaller than any element of the set Keys, and TAIL has to be bigger
21 |          $\forall k \in Keys : HEAD < k \wedge k < TAIL$ 
22 |-----|
  |
25 | TypeInvariant  $\triangleq$   mem stores all the nodes inserted into the list. It’s a function that assigns
26 |                       nodes to addresses. Each node as a key, a next field pointing to the address of
27 |                       the next node, and can be marked or not.
28 |                        $\wedge mem \in [Adr \rightarrow [key : Keys \cup \{0, 1, 100\},$ 
29 |                                   next :  $Adr \cup \{0\},$ 
30 |                                   mark :  $\{0, 1\}]]$ 
  |
32 |                       “setup” is 0 before the insertion of Head and Tail into the list, and 1 after that.
33 |                        $\wedge setup \in \{0, 1\}$ 
  |
35 |                       “proc” is a function of each process
36 |                        $\wedge proc \in [$ 
37 |                         Process  $\rightarrow [$  ninfo is a record that keeps track of the information regarding
38 |                         the current node to be inserted/deleted, like its key and position in the list.
39 |                         ninfo :  $[CNkey : Keys \cup \{0\}, CNnext : Adr \cup \{0\},$ 
40 |                               AdrLeft :  $Adr \cup \{0\}, AdrRight : Adr \cup \{0\},$ 
41 |                               RigNext :  $Adr \cup \{0\}],$ 
  |
43 |                       procIns and procDel state the finer-grained steps of the insert
44 |                       and delete actions
45 |                       procIns : {“readyI”, “createdI”, “locatedI”, “unique”,
46 |                                   “swapedI1”},
  |
48 |                       procDel : {“readyD”, “identifiedD”, “locatedD”,
49 |                                   “assignedD”, “swapedD1”},
  |
51 |                       A process can only start inserting/deleting a new node if it’s
52 |                       still free. Once it’s committed to a certain action it should carry it till the
53 |                       end. “choice” is used to represent that. A simple 2 bit variable, e.g. {“free”,
54 |                       “committed”}, would be a more elegant solution. The distinction between ‘insert’
55 |                       and ‘delete’ is necessary for ‘artificial reasons’ – see step “CreateI(p, key)”.
56 |                       choice : {“undecided”, “toinsert”, “todelete”}]

```

Figure 3: *Header* of the TLA<sup>+</sup> model

```

(*****)
(*          CONFIG HarrisR.cfg          *)
(*****)
SPECIFICATION Spec
(* This statement tells TLC that Spec is the specification to be
checked. *)
INVARIANTS TypeInvariant Coherence
(* This statement tells TLC to check that TypeInvariant and
Coherence are invariants of the specification. *)
(*****)
(* TLC requires that every declared constant in the specification *)
(*be assigned a value by a CONSTANT statement in the configuration*)
(*file. *)
(*****)
CONSTANT
Keys = {10,20,30,40,50}
Adr = {1201,1202,1203,1204,1205,1206,1207,1208}
Process = {p1,p2,p3,p4}
HEAD = 1
TAIL = 100
(***** End of config file *****)

```

Figure 4: Configuration file of Harris' Algorithm TLA<sup>+</sup> Model

```

324 |-----|
329  $Insert(i, k) \triangleq CreateI(i, k) \vee LocateI(i) \vee VerUniq(i) \vee CasI1(i) \vee CasI2(i)$ 
331  $Delete(i, k) \triangleq Identify(i, k) \vee LocateD(i) \vee AssignD(i) \vee CasD1(i) \vee CasD2(i)$ 
334  $Next \triangleq \vee SetInitNodes$ 
336  $\vee \exists i \in Process : \exists k \in Keys : Insert(i, k) \vee Delete(i, k)$ 
339  $Spec \triangleq Init \wedge \Box [Next]_{\langle mem, proc, setup \rangle}$ 
340 |-----|
341 THEOREM  $Spec \implies \Box (TypeInvariant \wedge Coherence)$ 
342 |-----|

```

Figure 5: Bottom of the TLA<sup>+</sup> Specification of Harris' Algorithm

```

59  $Coherence \triangleq$  "The key of the node that one node points to has to smaller than its own key"
60 LET set of all nodes pointing to another one
61  $nodp \triangleq \{j \in Adr : (mem[j].key \neq 0 \wedge mem[j].next \neq 0)\}$ 
62 IN Claim:
63  $\forall i \in nodp : mem[i].key < mem[mem[i].next].key$ 

```

Figure 6: Definition of *Coherence*

Insert		Delete	
<i>TLA<sup>+</sup> model</i>	<i>Text. desc.</i>	<i>TLA<sup>+</sup> model</i>	<i>Text. desc.</i>
CreateI	Ii	Identify	Id
LocateI	IIi	LocateD	IIId & IIIId
VerUniq	IIIi	AssignD	IVd
CasI1	IVi	CasD1	Vd
CasI2	Vi	CasD2	VIId

Figure 7: Correspondence of TLA<sup>+</sup> model with the textual description in section 1.2

by the conditions that the *setup* (insertion of *Head* and *Tail*) was already done:  $setup=1$  and that the process  $p$  is ready to start and not yet committed to any other action:  $proc[p].procIns="readyI" \wedge proc[p].choice="undecided"$ . An extra "artificial" guard is added, stating that there has to be free memory: the number of free memory locations is greater than the number of processes executions insertions. The function used here, *Cardinality* is defined in the standard module *FiniteSets*.

```

109 CreateI(p, key) ≜ ∧ setup = 1
110                   ∧ proc[p].procIns = "readyI"
111                   ∧ proc[p].choice = "undecided"
112                   Checks if there's still space in memmory
113                   ∧ (Cardinality({a ∈ Adr : mem[a].key = 0}) - Cardinality(
114                       {i ∈ Process : proc[i].choice = "toinsert"})) > 0
116                   ∧ proc' = [proc EXCEPT ![p].ninfo.CNkey = key,
117                               ![p].procIns = "created!",
118                               ![p].choice = "toinsert"]
119                   ∧ UNCHANGED ⟨mem, setup⟩

```

Figure 8: Definition of Action *CreateI*

The actualization of a variable in TLA<sup>+</sup> may be done in two different forms: (1) explicitly stating the values of all its fields, or (2) by saying that is the same as before, except for the fields listed after **EXCEPT**<sup>21</sup>. The actualizations in *CreateI* update *choice*, *procIns* and register the input parameter *key* as the key of the current node to insert.

The *LocateD(p)* action (Figure 9) identifies the left and right nodes and checks the *key* of the right node: if it's equal to the search key, right node is assigned to be deleted. Otherwise (the node that we want to delete is not in the linked-list) the process is set free ( $procDel=readyD$ ,  $choice=undecided$ ).

The definition of *LocateD* introduces the TLA<sup>+</sup> **LET/IN** construct. The **LET** clause consists of a sequence of definitions whose scope extends until the end of the **IN** clause. These local definitions can be used to shorten

<sup>21</sup>In our case, this feature is really useful since variable *proc* encapsulates many fields.

```

237  $LocateD(p) \triangleq \wedge proc[p].procDel = \text{"identifiedD"}$ 
239  $\wedge LET\ posr \triangleq \{j \in Adr : (mem[j].key \neq 0 \wedge mem[j].mark = 0 \wedge$ 
240  $mem[j].key \geq proc[p].ninfo.CNkey)\}$ 
242  $posl \triangleq \{j \in Adr : (mem[j].key \neq 0 \wedge mem[j].mark = 0 \wedge$ 
243  $mem[j].key < proc[p].ninfo.CNkey)\}$ 
245  $right \triangleq CHOOSE\ a \in posr : \forall i \in posr : mem[a].key \leq mem[i].key$ 
247  $left \triangleq CHOOSE\ a \in posl : \forall i \in posl : mem[a].key \geq mem[i].key$ 
249 IN
250 Check the identity of the right node: if its key is equal to the current one the process
251 continuous to execute and the right node is the node to be deleted; otherwise it aborts.
252 IF  $mem[right].key = proc[p].ninfo.CNkey$  THEN
254  $\wedge proc' = [proc\ EXCEPT\ ![p].ninfo.AdrLeft = left,$ 
255  $![p].ninfo.AdrRight = right,$ 
256  $![p].procDel = \text{"locatedD"}]$ 
257  $\wedge UNCHANGED\ \langle mem, setup \rangle$ 
259 ELSE abort (key doesn't exist)
260  $\wedge proc' = [proc\ EXCEPT\ ![p].ninfo.CNkey = 0,$ 
261  $![p].procDel = \text{"readyD"},$ 
262  $![p].choice = \text{"undecided"}]$ 
263  $\wedge UNCHANGED\ \langle mem, setup \rangle$ 

```

Figure 9: Definition of Action *LocateD*

expressions by replacing common subexpressions with an operator. It can also make an expression easier to read. Nested LETs are allowed and a good idea in large specifications.

Also introduced here is the operator **CHOOSE**. **CHOOSE**  $x:F$  equals an arbitrarily chosen value  $x$  that satisfies the formula  $F$ . Determinism is achieved if only one value of  $x$  satisfies  $F$ .

In *LocateD*, *right* is *chosen to be* the smallest element of *posr*, the set of all unmarked nodes whose key is greater or equal than the search key; *left* is *chosen to be* the highest element of *posl*, the set of all unmarked nodes whose key is smaller than the search key.

*CasD2(p)* (Figure 10) performs the *physical deletion* of the node from the linked-list. It is guarded by the pre-condition  $procDel=swapedD1$ , which represents that all the other intermediate actions have already been executed. It models the CAS operation: if the left node is still pointing to the right one  $mem[AdrLeft].next = AdrRight$ , it is made to point to the node identified as its immediate successor *RigNext*. Otherwise, it's sent back to the location phase ( $procDel = identifiedD$ ).

Note that there is no kind of *garbage collection*. The memory location is not erased. This is due to the reasons explained in section 1.4.

```

298  $CasD2(p) \triangleq \wedge proc[p].procDel = \text{"swapedD1"}$ 
300  $\wedge \text{IF } mem[proc[p].ninfo.AdrLeft].next = proc[p].ninfo.AdrRight \text{ THEN}$ 
302  $\wedge mem' = [mem \text{ EXCEPT } ![proc[p].ninfo.AdrLeft].next = proc[p].ninfo.RigNext]$ 
304  $\wedge proc' = [proc \text{ EXCEPT } ![p].ninfo.CNkey = 0,$ 
305  $![p].ninfo.CNnext = 0,$ 
306  $![p].ninfo.AdrLeft = 0,$ 
307  $![p].ninfo.AdrRight = 0,$ 
308  $![p].ninfo.RigNext = 0,$ 
309  $![p].procDel = \text{"readyD"},$ 
310  $![p].choice = \text{"undecided" } ]$ 
312  $\wedge \text{UNCHANGED } \langle setup \rangle$ 
314  $\text{ELSE } \text{search again}$ 
315  $\wedge proc' = [proc \text{ EXCEPT } ![p].ninfo.AdrLeft = 0,$ 
316  $![p].ninfo.AdrRight = 0,$ 
317  $![p].ninfo.RigNext = 0,$ 
318  $![p].procDel = \text{"identifiedD"}]$ 
319  $\wedge \text{UNCHANGED } \langle mem, setup \rangle$ 

```

Figure 10: Definition of Action *CasD2*

## 3 Promela and SPIN

### 3.1 The Language / Tool

Where as TLA<sup>+</sup> was first developed as a specification language on its own, and TLC came after as a model checker that could handle it, Promela is the specifically developed input language of SPIN, an on-the-fly explicit-state model checker (as TLC).

Promela resembles an imperative language like C augmented with a few communication primitives. Promela allows to describe the behavior of each process in a system, as well as the interactions between them. For communication, the processes may use FIFO communications channels, rendez-vous or shared variables.

SPIN does not permit to study infinite state systems. A key feature of SPIN is its availability of several state space reduction methods: state compression, on-the-fly verification and hashing techniques.

SPIN can check safety and liveness properties of a specification. When a property is violated, SPIN presents a counter-example.

### 3.2 Going in more detail - Overview of Promela

A Promela model is constructed from three basic types of objects:

- Processes
- Data objects

- Message channels

A process is identified by the keyword `proctype` and can have several instantiations. This can be done with the prefix `active [n]`, where  $n$  is the number of instantiations, or by using the operator `run`<sup>22</sup>. Example:

```
active 2 proctype ProcessA (parameter1)
```

The description of a system in Promela starts with constants definition and global variables declarations. Promela is a typed language and all variables must be declared before being referenced. There are two levels of scope: global and process local. The global variables can be accessed by all processes, functioning as the *shared memory* of the system<sup>23</sup>.

The types of variables are `bit`, `bool`, `byte`, `chan`, `mtype`, `pid`, `short`, `int` and `unsigned`.

From the provided basic data types, new ones can be created, in a scalable fashion, through `typedef`.

Message channels are used to model the exchange of data between processes. They are declared through `chan` and can be either local or global, as any other kind of variable. Similarly to as in CSP, `channame!` is used to specify the sent of a message through channel `channame`, and `channame?` the reception from the channel. Communication can be asynchronous or "rendezvous". The distinction is made through the size of the buffer of the channel: size zero defines a rendezvous port. A channel declaration has than the form

```
chan name = [buffer size] of { variable(s) type }
```

With `mtype` we can introduce variables that can hold symbolic values defined by the programmer (introduced with one or more `mtype` declarations). A limitation is that no distinction of different sets can be made, i.e., the separate declarations:

```
mtype = { data, control, error };
mtype = { chair, table, box };
```

are indistinguishable to SPIN to the single declaration:

```
mtype = { data, control, error, chair, table, box };
```

A central concept in Promela is **executability**. Every statement in a model is either *executable* or *blocked*, depending on the system state. This provides the basic means for modeling process synchronizations. Print and assignment statements are always executable. Booleans conditions are executable iff<sup>24</sup> they are true. Any statement that is non-executable can

---

<sup>22</sup>For differentiation each process is assigned a unique instantiation number. This is done automatically, the user doesn't need to worry about it. If necessary, each process can refer to its own instantiation number via the predefined local variable `_pid`.

<sup>23</sup>No intermediate levels can created, i.e. the scope of a global variable cannot be restricted to a subset of processes and the scope of a local one to specific blocks of statements.

<sup>24</sup>*if and only if*

block the executing process. Hence, if we want an action to happen only after certain condition(s) we can simply write:

```
(conditionA == true);  
/* description of the action */
```

For control flow, Promela supports the selection statement `if..fi` and the repetition loop `do..od`. Examples:

```
if  
:: (a != b) -> option1  
:: (a == b) -> option2  
fi  
do  
:: (a > b) -> a = a - b  
:: (a < b) -> b = b - a  
:: (a == b) -> break  
od
```

The element preceding `->` is the *guard* of the action<sup>25</sup>. Note that the guards need not to be mutually exclusive. If more than one guard is executable, one of the corresponding sequences will be selected nondeterministically.

SPIN can be used to check both safety and liveness properties of a system. The two main constructs to define the **correctness** requirements are *assertions* and *never claims*. An assertion statement has the form **assert** (*expression*) and is defined locally to a process, therefore checking the correctness of a system in a particular state. To define a *system invariant* – a property that should hold in every reachable state of the system, a **never** claim should be used.<sup>26</sup>

Some properties, like the absence of *deadlock* and *race conditions*, are so basic that are checked by default. Note that expressing correctness conditions through **never** claims, makes us work with *negated formulas*, i.e. if we want to express that a property *p* should always hold, what we actually write is that *non p* never occurs. To write a **never** claim that checks for the invariance of the system property *p* we can write:

```
never {  
  do  
  :: !p -> break  
  :: else  
  od  
}
```

A **never** claim can also be used to express properties in linear temporal logic (LTL). Strictly speaking, Promela does not include syntax for the

---

<sup>25</sup>This notation is inspired in [1].

<sup>26</sup>This is not mandatory though, for some properties it can be achieved with an assertion clause through the construction: `proctype monitor() { assert (invariant) }`, but this looks a "dirtier" solution.

specification of LTL formulas; SPIN has a separate parser that mechanically translates such formulas into Promela syntax, so that LTL can effectively become part of the language that is accepted by SPIN.

Another construct of Promela to deal with correctness claims are **labels**: By default a valid end state is one in which every process that was instantiated has reached the end of its code. However, if some process(es) doesn't reach this point, this may not be necessarily an error.<sup>27</sup> Therefore, to identify individual process states as valid end states the label **end** can be used. Similarly, there are also the labels **progress** and **accept**. The first to *validate* potentially infinite execution cycles, in the case that they visit the labeled state, and the later to mark states that should not be part of any potentially infinite execution cycle.

### 3.3 Model developed

The developed Promela model of Harris algorithm is now presented. Design decision and extracts of the model are explained. The full model is shown in Appendix B. The model was chronologically developed after (and made *as similar as reasonably advisable* to) the one written in TLA<sup>+</sup>. This description contains references to the later so it should be read after section 2.3.<sup>28</sup>

From the three Promela basic types of objects, message channels needed not to be used. Three processes and two global variables were used. The processes defined were *setup*, *insert* and *delete*. The last two contain all the operations necessary for the insertion/deletion of a node in/from the list; *setup* is an "artificial process" that inserts *Head* and *Tail* in the list.

The *main* global variable of the system is **mem**, defined as an array of nodes. **Node** is a user defined type of variable that contains three fields: *key*, *next* and *mark*. **setdone** is a boolean variable that equals true once *Head* and *Tail* are inserted in the list. Figure 11 shows the *header* of the model.

The processes in Promela are defined in a sequential manner. SPIN makes no assumption on the relative speed of processes execution and, when model-checking the system it automatically generates interleaving sequences of execution of the processes.<sup>29</sup> Though, along the processes specification there are comment lines referring to the equivalent TLA<sup>+</sup> action/textual description phase, e.g. `/* CreateI - Ii */`.<sup>30</sup> The names of the variables are also similar.

The correctness claim of the model — equivalent to *Coherence* (Figure 6, page 14), in TLA<sup>+</sup> — expresses that all the nodes registered in *mem* and

---

<sup>27</sup>It can be perfectly valid, for instance, for server processes to enter a wait state after a transition is completed, immune to the fact that all user processes have terminated.

<sup>28</sup>And after 3.2, naturally, if you are not familiar with Promela.

<sup>29</sup>Note that operations like reads and writes are executed atomically.

<sup>30</sup>See also Figure 7 on page 15.

```

#define L 4 /* Lenght of the memmory */
#define M 2 /* Number of inserting processes */
#define N 2 /* Number of deleting processes */
#define HEAD 1 /* Key value of the 'Head' */
#define TAIL 100 /* Key value of the 'Tail' */

typedef Node {
    byte key;
    byte next;
    bool mark
}
Node mem[L];
bool setdone=false

```

Figure 11: Header of Promela developed model

that point to another node, have a *key* that is strictly smaller than the key of the node that they point to. This was expressed in Promela as show in Figure 12.<sup>31</sup>

```

never {
    do
        :: !( !( (mem.key != 0) && (mem.next != 0) ) || (mem.key < mem[mem.next].key)) -> break
        :: else
    od
}

```

*Explanation:*

$(mem.key \neq 0) \ \&\& \ (mem.next \neq 0)$  — nodes registered that point to another node, *elegible*.

$(mem.key < mem[mem.next].key)$  — key strictly smaller than the key of the node that they point to, *keysmaller*.

We want to express: *elegible implies keysmaller, p*.

This is logically equivalent to: *not eligible or keysmaller, p*.

Final form comes from Promela syntax.

Figure 12: Never claim in the developed Promela model

`proctype insert()` starts (Figure 13) with the statement `(setdone==true)`. This serves as a synchronization method: an inserting processes can only start to execute after the *setup* is done. Till then it is blocked.

---

<sup>31</sup>Recall that a never claim that checks for the invariance of system property *p* can be written in Promela as:

```

never {
    do
        :: !p -> break
        :: else
    od
}

```

```

active [M] proctype insert() {
(setdone==true); /* Guarding condition for the executability of any inserting
process — the initial setup has to be finished first */

byte CNkey, CNnext, AdrLeft, AdrRight, t, t_next, left_next;
byte pos=0, counter=0, nonblank=0;
startinsert:
/* "Createl - li" */
atomic{
do /* count the elements in the list */
:: (mem[counter].key != 0 && mem[counter].key < TAIL) ->
nonblank ++;
counter = mem[counter].next;
:: else -> break
od;
if /* an insertion can only happen if there's space in mem */
::((L - nonblank - M) > 0) ->
if
:: CNkey=10
:: CNkey=20
:: CNkey=30
:: CNkey=40
:: CNkey=50
fi
:: else -> goto endinsert
fi }

```

Figure 13: Beginning of proctype insert()

The guard  $(L - \text{nonblank} - M) > 0$  expresses that an insertion can only take place if the memory is not full. The *generation* of the key to insert (`if ::CNkey=10 ... ::CNkey=50 fi`) is a bit *archaic*. Promela provides no predefined function to generate/select from a set a random number.

The *physical insertion* of the node (Figure 14) ends the `proctype`. The node is registered in `mem` in the lowest available blank position, `pos`. `atomic { ... }`, both in figure 13 and 14, is used to define a fragment of code that is to be executed indivisibly.

```

/* "Casl2 - Vi" */
atomic{
  do
    :: (mem[pos].key != 0) -> pos++
    :: else -> break
  od; }
if
  :: (mem[AdrLeft].next == AdrRight) ->
    mem[AdrLeft].next = pos;
    mem[pos].key = CNkey;
    mem[pos].next = CNnext
  :: else -> goto searchagain
fi;
endinsert:
  skip;
} /* End of proctype insert */

```

Figure 14: End of `proctype insert()`

The definition of `proctype delete()` brings no new aspects to this description. It starts with the statement (`setdone==true`), declaration of the local variables and the *archaic generation* of the node to delete. The search of the left and right nodes (Figure 15), in line with the others (`proctype insert()` and TLA<sup>+</sup> model), implements the simplification stated in the section 1.3. The removal of the node from the linked-list is also executed with no *garbage collection*.<sup>32</sup>

---

<sup>32</sup>See section 1.4.

```

/* LocateD - IIId */
searchagainD:
    t=0;
    t_next=mem[0].next;
    left_next=0;
    pos=0;
    do
    :: ((mem[t_next].mark==true) || (mem[t].key < CNkey)) ->
        if
        :: (mem[t_next].mark==false) ->
            AdrLeft = t;
            left_next = t_next;
        :: else ->skip
        fi;
        t=t_next;
        if
        :: (t==1) -> goto endcycleD
        :: else ->skip
        fi;
        t_next = mem[t].next
    :: else -> break
    od;
endcycleD:
    AdrRight=t;
    if
    :: (left_next == AdrRight) ->
        if
        :: ((AdrRight!=1) && (mem[mem[AdrRight].next].mark == true)) ->
            goto searchagainD
        :: else -> goto endsearchD
        fi
    :: else -> goto endsearchD
    fi;
endsearchD:
    skip;
/* Examine - IIIId */
    if
    :: ((AdrRight!=1) || (mem[AdrRight].key != CNkey)) -> goto enddelete
    :: else -> skip
    fi;

```

Figure 15: Identification of the left and right nodes

## 4 Comparison

Naturally, both formalisms have strengths and weaknesses over the other. To make a comparison as objective as possible, several *measurable* comparison criteria were defined.<sup>33</sup> These criteria were divided in three categories, according to their scope: global, specific of the tool — SPIN or TLC — and specific of the language — Promela or TLA<sup>+</sup>.

Figure 16 summarizes the evaluation. For every criteria, a *classification* from 0 (minimum) to 4 (maximum) was given to the systems.<sup>34</sup> Note that simply adding the classifications shouldn't be the basis for a decision, some criteria may have an importance considerably bigger/smaller than other, according to the intended application.

### 1.1 Matching of the method to the application

Once both systems are Model-Checkers they have many aspects in common. They are both appropriate to analyze concurrent systems and both have support for mechanisms as the expression of properties in temporal logic.

### 1.2 Human factors

Compared to other Formal approaches that imply the writing of proofs, Model-Checking can be considered of easier use. (Even though, the necessity for learning and training is, certainly, not negligible.) Because a significant part of the job is done automatically by the tool, Model Checkers should be more *productive* than techniques as theorem proving.

### 1.3 Widespread utilization

Model checkers have achieved a relatively good popularity as a technique for the analysis and verification of concurrent systems. TLA<sup>+</sup> / TLC has been used in the design and validation of protocols at places like Compaq, Intel and, more recently, Microsoft. SPIN is, however, a more widespread tool, definitely a very popular one in the field. One factor that justifies this is that TLC considerably younger than SPIN.

### 2.1 Licensing / Distribution

Both tools are excellent in this aspect. They require no license fee and can be freely downloaded from the internet. They can also both run in Windows and Unix systems. (SPIN also in Mac).

### 2.2 Maturity

Both tools keep being upgraded continuously. TLC (from 1999) is, however, considerably younger than SPIN (from 1991).

### 2.3 Performance

Both tools have similar limitations: they are both explicit state model checkers and cannot handle with infinite systems. All the variables have to

---

<sup>33</sup>[19] served as a good starting point for this exercise.

<sup>34</sup>In the figure,  $\square$  is used if both systems are given an equivalent classification and  $\diamond$  otherwise.

COMPARISON CRITERIA			Promela / SPIN	TLA+ / TLC	
1.	GLOBAL	1.1 Matching of the method to the application	□ □ □	□ □ □	
		1.2 Human factors	1.2.1 Ease of use	□ □ □	□ □ □
			1.2.2 Productivity	□ □	□ □
		1.3 Widespread utilization	◇ ◇ ◇	◇ ◇	
2.	TOOL	2.1 Licensing / Distribution	□ □ □ □	□ □ □ □	
		2.2 Maturity	◇ ◇ ◇	◇	
		2.3 Performance	2.3.1 State space	□ □	□ □
			2.3.2 Speed	◇ ◇ ◇	◇
		2.4 Interface	□ □	□ □	
		2.5 Coverage of the input language	□ □ □	□ □ □	
3.	LANGUAGE	3.1 Bibliography	3.1.1 Availability	◇ ◇ ◇	◇ ◇ ◇ ◇
			3.1.2 Quality	◇ ◇	◇ ◇ ◇ ◇
		3.2 Expressiveness	◇	◇ ◇ ◇ ◇	
		3.3 Readability	◇	◇ ◇ ◇ ◇	
		3.4 Reusability	◇	◇ ◇ ◇	
		3.5 Scalability / evolutivity	◇	◇ ◇ ◇	
		3.6 Level of abstraction	□ □ □	□ □ □	
		3.7 Checking possibilities	□ □ □	□ □ □	
		3.8 Lifecycle coverage	3.8.1 Requirements	-----	-----
			3.8.2 Specification	◇ ◇	◇ ◇ ◇ ◇
3.8.3 Implementation	◇ ◇ ◇		-----		

Figure 16: Comparison of the formalisms

be bound. Because TLC is written in Java it is slower than SPIN, whose verifications are executed with a C compiled file.

## 2.4 Interface

The interfaces of both SPIN and TLC are generally simple. They both run in batch mode and, when finding an error, produce a counter example reasonably understandable. SPIN as a very simple graphic interface.

## 2.5 Coverage of the input language

Promela is the specifically developed input language of SPIN. Therefore it is fully covered by it. On the other hand, TLA<sup>+</sup> was first developed than TLC, as a high-level specification language. Because TLA<sup>+</sup> has such high-level capabilities, TLC cannot handle the all of it. This is though not a practical problem, TLC handles all the specifications that "arise in describing actual systems" [10].

## 3.1 Bibliography

Both languages have a reference manual, written by their developers: [6] for Promela and [12] for TLA<sup>+</sup>. [12] can, however, be freely downloaded from the internet.<sup>35</sup>

[12] is also an excellently written book. It presents TLA<sup>+</sup> with a series of complexity growing examples and takes a great deal in making the reader understand the principles of the language, it enlightens the fact that TLA<sup>+</sup> provides a nice way to formalize the style of reasoning about systems. In [18] one can found the quote: In order to use these tools effectively you need a good grasp of the fundamentals of CSP: the tools are certainly not and *alternative* to gaining an understanding of the theory. Therefore this book is still, in the first instance, a text on the principles of the language rather than being a manual on how to apply its tools. [6] is not so well succeeded in this. Also, several dispensable references to similarities with C are constantly present, as well as "dubious alternative solutions" to problems with a simple clean one.<sup>36</sup>

## 3.2 Expressiveness

TLA<sup>+</sup> is clearly more expressive than Promela. It guides us to think logically reason about the system to develop and everything can be expressed as *what we want* and not on *how to get there*, capability very useful when specifying systems. Constructions like `do .. od` loops to compute the value of a variable need not to be used, they can be more simply expressed as a function. Recall, for instance, the use of `Cardinality` in `CreateI`, the definitions of `left` and `right` in `LocateD`, or the *archaic generation* of a node in `proctype insert()`.

Another example could be the correctness claims of the models. The expression of *Coherence* — Figure 6, is more natural and easier to devise its equivalent `never` claim in Promela — Figure 12. As an *historical note* it can be left here the note that the later was considerably hard to reach. In

---

<sup>35</sup><http://research.microsoft.com/users/lamport/tla/book.html>

<sup>36</sup>As a curiosity, you can, for instance, see page 51.

a first approach and *brainwashed* by the C like look of Promela the result result had been:

```

never {
do
:: (mem[index].next != 0) ->
    if
        :: (mem[index].key < mem[mem[index].next].key) ->
            skip;
            index = mem[index].next;
        :: else ->
            index=0;
            wrong=true;
            break
    fi
:: wrong -> break
:: else
od
}

```

### 3.3 Readability

A model of reasonable dimension is more easily read in TLA<sup>+</sup> than in Promela. To understand the behavior of a system modeled in TLA<sup>+</sup> can be achieved by reading its *header* and than *bottom* of the module. Furthermore, the automatically generated L<sup>A</sup>T<sub>E</sub>X documents of the models give them a really *nice and clean* look.

One may be refuting this idea by the aversion to mathematical symbols. This may be natural because of some unfamiliarity and lack of use in dealing with them, but all it takes is that initial step. Bellow an example on how powerful it can be.

Take a lights system. One property about them is that *every time that the light is yellow, it eventually turns red*. In an LTL formula this is expressed as:

$$\Box((L = \textit{yellow}) \Rightarrow \Diamond(L = \textit{red}))$$

It should take no more than being told that  $\Box$  reads *always*,  $\Rightarrow$  reads *implies* and  $\Diamond$  *eventually*, to understand the previous and similar expressions.

Now, recall that Promela does not include syntax for the specification of LTL formulas — SPIN has a separate parser that mechanically translates such formulas into Promela syntax. Abbreviating the formula as:

$$\Box(p \Rightarrow \Diamond q)$$

and running this parser produces the result:

```

never { /* [] (p -> <> q) */
T0_init:
    if
        :: ((! (p)) || (q)) -> goto accept_S20
        :: (1) -> goto T0_S27
    fi
}

```

```

        fi;
accept_S20:
    if
        :: ((! (p)) || (q)) -> goto T0_init
        :: (1) -> goto T0_S27
    fi;
accept_S27:
    if
        :: (q) -> goto T0_init
        :: (1) -> goto T0_S27
    fi;
T0_S27:
    if
        :: (q) -> goto accept_S20
        :: (1) -> goto T0_S27
        :: (q) -> goto accept_S27
    fi;
}

```

free of the LTL operators but *practicably unreadable*. Do please note that this is not a criticism to Promela/SPIN — what we want to debate is the aversion that one can have to mathematical symbols. The shown result is fully automatically generated by SPIN, by simply typing: `spin -f "[ (p -> <> q) ]"` in the command line, and the *readable* original LTL formula is shown as a comment (between `/* */`) at the beginning.

### 3.4 Reusability

Reusability is an issue in computer science. It is achieved by the possibility of creating modular, parametric or generic descriptions. TLA<sup>+</sup> incorporates this features significantly better than Promela. A specification is easily spread over different modules, that are than related with the commands EXTENDS and INSTANCE. This is not possible in Promela.

### 3.5 Scalability

This concerns the possibility of modifying or extending descriptions in an incremental manner. It is also related to modular, parametric and generic descriptions.

### 3.6 Level of abstraction

In this aspect we can say that TLA<sup>+</sup> and Promela have *equivalent capabilities*. It is important to be able to specify a system a different levels of abstraction. TLA<sup>+</sup> can go to a higher-level than Promela and, in turn, Promela can go to a level closer to the final implementation solution.

### 3.7 Checking possibilities

Promela and TLA<sup>+</sup> have similar capabilities of expressing / checking correctness claims in combination with their tools, SPIN and TLC.

### 3.8 Coverage of lifecycle

None of the languages is appropriate to express requirements.

TLA<sup>+</sup> is a specification language by excellence. The aspects reported before certificate that.

The proximity of a language to the final implementation may be a very important aspect, depending on the use that we want to do. Promela clearly overcomes TLA<sup>+</sup> in this aspect. TLA<sup>+</sup> is best suited for reasoning about protocols, whereas a Promela model can be *close* to the final implementation solution.

### **Last Remarks**

In this analysis of advantages/disadvantages of the tools, or the languages, or both over each other, one point should be a basis for a decision: more focus should be given to the well-matched characteristics of the languages with the object of modeling, than to the performance of the tools, provided that the later are not *unreasonably different*.

To validate a model of a system, great deal should be given to the validity of the abstraction decisions introduced in the model. The tool is a support to perform the checks.

If the application is related to the analysis of *higher level aspects*, then TLA<sup>+</sup> clearly overcomes Promela. The lowest is the level of the solution we want to validate, the hardest will the decision be.

# A TLA<sup>+</sup> model of Harris' Algorithm

```

1 |----- MODULE HarrisR -----|
  This model represents the algorithm presented by Harris for the implementation of non-blocking linked-lists.
  In this model the nodes inserted to the list are stored in the global variable "mem". The actions of deleting and inserting
  nodes to the list are divided, for a finer-grained solution in several intermediate steps.
  For 'artificial' (to be seen in some comments bellow) we mean all the aspects that are not directly connected to the
  algorithm, but that represent adaptations of it to be possible/more easily modeled.
10 EXTENDS Naturals, Sequences, FiniteSets
11 CONSTANT Adr,           the set of addresses
12           Keys,         the set of keys
13           Process,      the set of processes
14           HEAD, TAIL  values of the keys of 'Head' and 'Tail'

16 VARIABLES mem,        'state of the memory', with all the nodes inserted in the Linked List
17           proc,        auxiliary variable – information for the intermediate stages
18           setup        'artificial' variable for the initial insertion of 'Head' and 'Tail'

20 ASSUME  HEAD has to be smaller than any element of the set Keys, and TAIL has to be bigger
21          $\forall k \in Keys : HEAD < k \wedge k < TAIL$ 
22 |-----|

25 TypeInvariant  $\triangleq$   mem stores all the nodes inserted into the list. It's a function that assigns
26                       nodes to addresses. Each node as a key, a next field pointing to the adress of
27                       the next node, and can be marked or not.
28                        $\wedge mem \in [Adr \rightarrow [key : Keys \cup \{0, 1, 100\},$ 
29                                   next :  $Adr \cup \{0\}$ ,
30                                   mark :  $\{0, 1\}]]$ 

32                       "setup" is 0 before the insertion of Head and Tail into the list, and 1 after that.
33                        $\wedge setup \in \{0, 1\}$ 

35                       "proc" is a function of each process
36                        $\wedge proc \in [$ 
37                         Process  $\rightarrow [$  ninfo is a record that keeps track of the information regarding
38                         the current node to be inserted/deleted, like its key and position in the list.
39                         ninfo :  $[CNkey : Keys \cup \{0\}, CNnext : Adr \cup \{0\},$ 
40                               AdrLeft :  $Adr \cup \{0\}, AdrRight : Adr \cup \{0\},$ 
41                               RigNext :  $Adr \cup \{0\}]$ ,

43                       procIns and procDel state the finer-grained steps of the insert
44                       and delete actions
45                       procIns : {"readyI", "createdI", "locatedI", "unique",
46                                   "swapedI1"},

48                       procDel : {"readyD", "identifiedD", "locatedD",
49                                   "assignedD", "swapedD1"},

51                       A process can only start inserting/deleting a new node if it's
52                       still free. Once it's committed to a certain action it should carry it till the
53                       end. "choice" is used to represent that. A simple 2 bit variable, e.g. {"free",
54                       "committed"}, would be a more elegant solution. The distinction between 'insert'
55                       and 'delete' is necessary for 'artificial reasons' – see step "CreateI(p, key)".
56                       choice : {"undecided", "toinsert", "todelete"}]]

```



```

127      $\wedge$  LET  $elemr \triangleq \{j \in \text{Adr} : (\text{mem}[j].\text{key} \neq 0 \wedge \text{mem}[j].\text{mark} = 0 \wedge$ 
128          $\text{mem}[j].\text{key} \geq \text{proc}[p].\text{ninfo}.\text{CNkey})\}$ 
130      $eleml \triangleq \{j \in \text{Adr} : (\text{mem}[j].\text{key} \neq 0 \wedge \text{mem}[j].\text{mark} = 0 \wedge$ 
131          $\text{mem}[j].\text{key} < \text{proc}[p].\text{ninfo}.\text{CNkey})\}$ 
134      $right \triangleq$  CHOOSE  $a \in elemr : \forall i \in elemr : \text{mem}[a].\text{key} \leq \text{mem}[i].\text{key}$ 
136      $left \triangleq$  CHOOSE  $a \in elemr : \forall i \in elemr : \text{mem}[a].\text{key} \geq \text{mem}[i].\text{key}$ 
139      $adj \triangleq \text{mem}[left].\text{next} = right$  Check if nodes are adjacent
141     IN
142     IF  $adj = \text{TRUE}$  THEN proceed
144          $\wedge \text{proc}' = [\text{proc} \text{ EXCEPT } ![p].\text{ninfo}.\text{AdrLeft} = left,$ 
145              $![p].\text{ninfo}.\text{AdrRight} = right,$ 
146              $![p].\text{procIns} = \text{"located"}]$ 
147          $\wedge \text{UNCHANGED} \langle \text{mem}, \text{setup} \rangle$ 
149     ELSE search again
150          $\wedge \text{proc}' = [\text{proc} \text{ EXCEPT } ![p].\text{procIns} = \text{"created"}]$ 
151          $\wedge \text{UNCHANGED} \langle \text{mem}, \text{setup} \rangle$ 

```

'Verification of uniqueness' – Checks if the node already exists in the list. This is the case when the key value of the right node is the same as the key value of the node to be inserted

```

158      $VerUniq(p) \triangleq \wedge \text{proc}[p].\text{procIns} = \text{"located"}$ 
160      $\wedge$  IF  $\text{mem}[\text{proc}[p].\text{ninfo}.\text{AdrRight}].\text{key} \neq \text{proc}[p].\text{ninfo}.\text{CNkey}$  THEN
162          $\text{proc}' = [\text{proc} \text{ EXCEPT } ![p].\text{procIns} = \text{"unique"}]$  proceed
164     ELSE If the key already exists, the process is aborted
165          $\text{proc}' = [\text{proc} \text{ EXCEPT } ![p].\text{procIns} = \text{"ready"},$ 
166              $![p].\text{choice} = \text{"undecided"}]$ 
168      $\wedge \text{UNCHANGED} \langle \text{mem}, \text{setup} \rangle$ 

```

The current node is made to point to the right node:

```

173      $CasI1(p) \triangleq \wedge \text{proc}[p].\text{procIns} = \text{"unique"}$ 
174      $\wedge \text{proc}' = [\text{proc} \text{ EXCEPT } ![p].\text{ninfo}.\text{CNnext} = \text{proc}[p].\text{ninfo}.\text{AdrRight},$ 
175          $![p].\text{procIns} = \text{"swaped1"}]$ 
176      $\wedge \text{UNCHANGED} \langle \text{mem}, \text{setup} \rangle$ 

```

The physical insertion of the node, recurring to CAS (addr, old, new), is attempted: the node that is being presently pointed at that instant by the left node [addr] is compared to the one previously identified [old], whose value has been stored in a variable and :

- If they match, the left node is made to point to the current node, concluding the inserting process.
- If they don't match (which means that some change in the list has been made in the mean time), the process goes back to `LocateI`.

```

185      $CasI2(p) \triangleq \wedge \text{proc}[p].\text{procIns} = \text{"swaped1"}$ 
187      $\wedge$  IF  $\text{mem}[\text{proc}[p].\text{ninfo}.\text{AdrLeft}].\text{next} = \text{proc}[p].\text{ninfo}.\text{AdrRight}$  THEN
190         LET  $pos \triangleq$  CHOOSE  $a \in \{b \in \text{Adr} : \text{mem}[b].\text{key} = 0\} :$ 
191              $\forall i \in \{j \in \text{Adr} : \text{mem}[j].\text{key} = 0\} : a \leq i$ 
192     IN insert in mem

```

```

193       $\wedge mem' = [mem \text{ EXCEPT } ![pos].key = proc[p].ninfo.CNkey,$ 
194       $![pos].next = proc[p].ninfo.CNnext,$ 
195       $![proc[p].ninfo.AdrLeft].next = pos]$ 

197      and Reset the procedure
198       $\wedge proc' = [proc \text{ EXCEPT } ![p].ninfo.CNkey = 0,$ 
199       $![p].ninfo.CNnext = 0,$ 
200       $![p].ninfo.AdrLeft = 0,$ 
201       $![p].ninfo.AdrRight = 0,$ 
202       $![p].ninfo.RigNext = 0,$ 
203       $![p].procIns = \text{"ready!"},$ 
204       $![p].choice = \text{"undecided"}]$ 

206       $\wedge \text{UNCHANGED } \langle setup \rangle$ 

209      ELSE search again
211       $\wedge proc' = [proc \text{ EXCEPT } ![p].procIns = \text{"created!"}]$ 
213       $\wedge \text{UNCHANGED } \langle mem, setup \rangle$ 

```

End of the insertion process

```

219 |-----|

```

This is the beginning of the 'Deletion process'. It's divided in 5 steps: 'Identify', 'LocateD', 'AssignD', 'CasD1' and 'CasD2'.

'Creation of the node to insert' – an identifier key is selected

```

226  $Identify(p, key) \triangleq \wedge setup = 1$ 
227       $\wedge proc[p].procDel = \text{"readyD"}$ 
228       $\wedge proc[p].choice = \text{"undecided"}$ 
229       $\wedge proc' = [proc \text{ EXCEPT } ![p].ninfo.CNkey = key,$ 
230       $![p].procDel = \text{"identifiedD"},$ 
231       $![p].choice = \text{"todelete"}]$ 
232       $\wedge \text{UNCHANGED } \langle mem, setup \rangle$ 

```

The list is searched in order to identify the left and right nodes: the left is the unmarked node that has the biggest key strictly smaller; the right is the unmarked node that has the smallest key greater or equal (than the current's node key).

```

237  $LocateD(p) \triangleq \wedge proc[p].procDel = \text{"identifiedD"}$ 

239       $\wedge \text{LET } posr \triangleq \{j \in Adr : (mem[j].key \neq 0 \wedge mem[j].mark = 0 \wedge$ 
240       $mem[j].key \geq proc[p].ninfo.CNkey)\}$ 

242       $posl \triangleq \{j \in Adr : (mem[j].key \neq 0 \wedge mem[j].mark = 0 \wedge$ 
243       $mem[j].key < proc[p].ninfo.CNkey)\}$ 

245       $right \triangleq \text{CHOOSE } a \in posr : \forall i \in posr : mem[a].key \leq mem[i].key$ 
247       $left \triangleq \text{CHOOSE } a \in posl : \forall i \in posl : mem[a].key \geq mem[i].key$ 

249      IN
250      Check the identity of the right node: if its key is equal to the current one the process
251      continuous to execute and the right node is the node to be deleted; otherwise it aborts.
252      IF  $mem[right].key = proc[p].ninfo.CNkey$  THEN

254       $\wedge proc' = [proc \text{ EXCEPT } ![p].ninfo.AdrLeft = left,$ 
255       $![p].ninfo.AdrRight = right,$ 
256       $![p].procDel = \text{"locatedD"}]$ 
257       $\wedge \text{UNCHANGED } \langle mem, setup \rangle$ 

```

```

259         ELSE abort (key doesn't exist)
260              $\wedge$   $proc' = [proc \text{ EXCEPT } ![p].ninfo.CNkey = 0,$ 
261                  $![p].procDel = \text{"readyD"},$ 
262                  $![p].choice = \text{"undecided"}]$ 
263              $\wedge$  UNCHANGED  $\langle mem, setup \rangle$ 

```

The immediate successor of the right node is stored in 'RigNext'

```

268  $AssignD(p) \triangleq \wedge$   $proc[p].procDel = \text{"locatedD"}$ 
269      $\wedge$   $proc' = [proc \text{ EXCEPT } ![p].ninfo.RigNext = mem[proc[p].ninfo.AdrRight].next,$ 
270          $![p].procDel = \text{"assignedD"}]$ 
271      $\wedge$  UNCHANGED  $\langle mem, setup \rangle$ 

```

Marking of the node: if the right node next field is still pointing to the node previously identified, then the node is marked. If not the process goes back to 'LocatedD'.

```

276  $CasD1(p) \triangleq \wedge$   $proc[p].procDel = \text{"assignedD"}$ 
277     The operation is guarded by the pre-condition that RigNext is not a marked node; otherwise
278     the process goes directly back to 'LocatedD'
279      $\wedge$  IF  $mem[proc[p].ninfo.RigNext].mark = 0$  THEN
281         IF  $mem[proc[p].ninfo.AdrRight].next = proc[p].ninfo.RigNext$  THEN
283              $\wedge$   $mem' = [mem \text{ EXCEPT } ![proc[p].ninfo.AdrRight].mark = 1]$ 
284              $\wedge$   $proc' = [proc \text{ EXCEPT } ![p].procDel = \text{"swapedD1"}]$ 
285              $\wedge$  UNCHANGED  $\langle setup \rangle$ 
287         ELSE search again
288              $\wedge$   $proc' = [proc \text{ EXCEPT } ![p].procDel = \text{"identifiedD"}]$ 
289              $\wedge$  UNCHANGED  $\langle mem, setup \rangle$ 
291     ELSE search again
292          $\wedge$   $proc' = [proc \text{ EXCEPT } ![p].procDel = \text{"identifiedD"}]$ 
293          $\wedge$  UNCHANGED  $\langle mem, setup \rangle$ 

```

Removal of the node from the linked-list: if the left node next field is still pointing to the right node, then it is made to point to the node identified as its immediate successor. If the comparison fails, the process goes back to 'LocatedD'.

```

298  $CasD2(p) \triangleq \wedge$   $proc[p].procDel = \text{"swapedD1"}$ 
300      $\wedge$  IF  $mem[proc[p].ninfo.AdrLeft].next = proc[p].ninfo.AdrRight$  THEN
302          $\wedge$   $mem' = [mem \text{ EXCEPT } ![proc[p].ninfo.AdrLeft].next = proc[p].ninfo.RigNext]$ 
304          $\wedge$   $proc' = [proc \text{ EXCEPT } ![p].ninfo.CNkey = 0,$ 
305              $![p].ninfo.CNnext = 0,$ 
306              $![p].ninfo.AdrLeft = 0,$ 
307              $![p].ninfo.AdrRight = 0,$ 
308              $![p].ninfo.RigNext = 0,$ 
309              $![p].procDel = \text{"readyD"},$ 
310              $![p].choice = \text{"undecided"}]$ 
312          $\wedge$  UNCHANGED  $\langle setup \rangle$ 
314     ELSE search again
315          $\wedge$   $proc' = [proc \text{ EXCEPT } ![p].ninfo.AdrLeft = 0,$ 
316              $![p].ninfo.AdrRight = 0,$ 
317              $![p].ninfo.RigNext = 0,$ 
318              $![p].procDel = \text{"identifiedD"}]$ 
319          $\wedge$  UNCHANGED  $\langle mem, setup \rangle$ 

```

End of the deletion process

324 |-----|

“Summary” of the specification

329  $Insert(i, k) \triangleq CreateI(i, k) \vee LocateI(i) \vee VerUniq(i) \vee CasI1(i) \vee CasI2(i)$

331  $Delete(i, k) \triangleq Identify(i, k) \vee LocateD(i) \vee AssignD(i) \vee CasD1(i) \vee CasD2(i)$

334  $Next \triangleq \vee SetInitNodes$

336  $\vee \exists i \in Process : \exists k \in Keys : Insert(i, k) \vee Delete(i, k)$

339  $Spec \triangleq Init \wedge \square[Next]_{\langle mem, proc, setup \rangle}$

340 |-----|

341 THEOREM  $Spec \implies \square(TypeInvariant \wedge Coherence)$

342 |-----|

## B Promela model of Harris' Algorithm

```

#define L 4 /* Length of the memory */
#define M 2 /* Number of inserting processes */
#define N 2 /* Number of deleting processes */
#define HEAD 1 /* Key value of the 'Head' */
#define TAIL 100 /* Key value of the 'Tail' */

typedef Node {
    byte key;
    byte next;
    bool mark;
}
Node mem[L];
bool setdone=false

/*=====|=====|=====|=====|=====| */

/* setup - Insertion of 'Head' and 'Tail' */

active proctype setup()
{
    if
        ::setdone == false ->
            atomic{
                mem[0].key=HEAD;
                mem[0].next=1;
                mem[1].key=TAIL;
                setdone=true;
            }
        ::else->skip
    fi
}

/* =====|=====|=====|=====|=====|

                                Insert process

=====|=====|=====|=====|=====| */

active [M] proctype insert()
{
    (setdone==true); /* Guarding condition for the executability of any inserting process - the
initial setup has to be finished first */

    byte CNkey, CNnext, AdrLeft, AdrRight, t, t_next, left_next;
    byte pos=0, counter=0, nonblank=0;

startinsert:
/* "Createl - li" */
atomic{
    do /* count the elements in the list */
        :: (mem[counter].key != 0 && mem[counter].key < TAIL) ->
            nonblank ++;
            counter = mem[counter].next;
        :: else -> break

```

```

od;

if /* an insertion can only happen if there's space in mem */
::((L - nonblank - M) > 0) ->
  if
  :: CNkey=10
  :: CNkey=20
  :: CNkey=30
  :: CNkey=40
  :: CNkey=50
  fi
  :: else -> goto endinsert
fi
}

/* "Locatel - III" */
searchagain:
  t=0;
  t_next=mem[0].next;
  left_next=0;
  pos=0;

  do
  :: ((mem[t_next].mark==true) || (mem[t].key < CNkey)) ->
  if
  :: (mem[t_next].mark==false) ->
  AdrLeft = t;
  left_next = t_next;
  :: else ->skip
  fi;
  t=t_next;
  if
  :: (t==1) -> goto endcycle
  :: else ->skip
  fi;
  t_next = mem[t].next;
  :: else -> break
od;

endcycle:
  AdrRight=t;

  if
  :: (left_next == AdrRight) ->
  if
  :: ((AdrRight!=1) && (mem[mem[AdrRight].next].mark == true)) ->
  goto searchagain
  :: else -> goto endsearch
  fi
  :: else -> goto endsearch
fi;

endsearch:
  skip;

/* "VerUniq - III" */

```

```

    if
    :: ((AdrRight!=1) && (mem[AdrRight].key == CNkey)) -> goto endinsert
    :: else -> skip
    fi;

/* "Cas1 - IVi" */
    CNnext = AdrRight;

/* "Cas2 - VI" */
atomic{
    do
    :: (mem[pos].key != 0) -> pos++
    :: else -> break
    od;
}

    if
    :: (mem[AdrLeft].next == AdrRight) ->
        mem[AdrLeft].next = pos;
        mem[pos].key = CNkey;
        mem[pos].next = CNnext
    :: else -> goto searchagain
    fi;
endinsert:
    skip;

} /* End of proctype insert */

/* =====|=====|=====|=====|=====|
                                Delete process
=====|=====|=====|=====|=====| */

active [N] proctype delete()
{
(setdone==true); /* Guarding condition for the executability of any deleting process - the
initial setup has to be finished first */

byte CNkey, AdrLeft, AdrRight, RighNext, pos;
byte t=0;
byte t_next=mem[0].next;
byte left_next=0;

/* Identify - Id */
    if
    :: CNkey=10
    :: CNkey=20
    :: CNkey=30
    :: CNkey=40
    :: CNkey=50
    fi;

/* Located - IId */
searchagainD:
    t=0;

```

```

t_next=mem[0].next;
left_next=0;
pos=0;

do
:: ((mem[t_next].mark==true) || (mem[t].key < CNkey)) ->
  if
  :: (mem[t_next].mark==false) ->
    AdrLeft = t;
    left_next = t_next;
  :: else ->skip
  fi;
  t=t_next;
  if
  :: (t==1) -> goto endcycleD
  :: else ->skip
  fi;
  t_next = mem[t].next
:: else -> break
od;

endcycleD:
  AdrRight=t;

  if
  :: (left_next == AdrRight) ->
    if
    :: ((AdrRight!=1) && (mem[mem[AdrRight].next].mark == true)) ->
      goto searchagainD
    :: else -> goto endsearchD
    fi
  :: else -> goto endsearchD
  fi;

endsearchD:
  skip;

/* Examine - IIIId */
  if
  :: ((AdrRight!=1) || (mem[AdrRight].key != CNkey)) -> goto enddelete
  :: else -> skip
  fi;

/* AssignD - IVd */
  RigNext = mem[AdrRight].next;

/* CasD1 - Vd */
  if
  :: (mem[RigNext].mark == false) ->
    if
    :: (mem[AdrRight].next == RigNext) -> mem[AdrRight].mark = true
    :: else -> goto searchagainD
    fi;
  :: else -> goto searchagainD
  fi;

```

```

/* CasD2 - VId */
  if
  :: (mem[AdrLeft].next == AdrRight) -> mem[AdrLeft].next == RigNext
  :: else -> goto searchagainD
  fi;

enddelete:
  skip;

} /* End of proctype delete */

/* =====|=====|=====|=====|=====|
                                     Correctness Claim
=====|=====|=====|=====|=====| */

never {
  do
  :: !( !( (mem.key != 0) && (mem.next != 0) ) || (mem.key < mem[mem.next].key)) ->
break
  :: else
  od
}
/* End of model */

```

## References

- [1] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
- [2] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.
- [3] Hui Gao and Wim H. Hesselink. A general lock-free algorithm using compare-and-swap, October 2004.
- [4] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, London, UK, 2001. Springer-Verlag.
- [5] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.
- [6] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Pearson Education, 2003.
- [7] IBM. IBM System/370 Extended Architecture, Principles of Operation. Publication No. SA22-7085, 1983.
- [8] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual. Volume 1: Application Architecture*. 2002, Revision 2.1.
- [9] Prasad Jayanti and Srdjan Petrovic. Efficient Wait-Free Implementation of Multiword LL/SC Variables. Technical Report TR2004-523, Dartmouth College, Computer Science, Hanover, NH, 2004.
- [10] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and verifying systems with TLA+, 2002.
- [11] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
- [12] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [13] Maged Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit cas. In Rachid Guerraoui, editor, *Distributed algorithms*, volume 3274/2004 of *Lecture Notes in Computer Science*, pages 144–158, Oct 2004.
- [14] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM Press.
- [15] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30, New York, NY, USA, 2002. ACM Press.
- [16] Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC23089, IBM Research Division, 2004.
- [17] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228, New York, NY, USA, 1997. ACM Press.
- [18] A. W. Roscoe. *The Theory and Practice of Concurrency*. Pearson Education Limited, Essex, CM20 2JE, England, 1998.
- [19] Joseph Sifakis. Formal methods and their evaluation. [www-verimag.imag.fr/~sifakis/RECH/FEMSYS/paper.ps](http://www-verimag.imag.fr/~sifakis/RECH/FEMSYS/paper.ps).
- [20] SPARC International. *The SPARC architecture manual: Version 9*. Prentice-Hall, 1994.

- [21] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66, London, UK, 1999. Springer-Verlag.