

System Development through Refinement

A First Acquaintance with the B Method

INTERNAL REPORT – DRAFT VERSION

Abstract

B is a software development method which is based on the concepts of weakest preconditions and refinement calculus. Safety properties of concurrent systems can be expressed in an extension of the B language called Event B. We show how we used the B method to proof the correctness of Harris' non-blocking linked-list algorithm and we discuss the suitability of (Event) B for the specification and verification of concurrent systems.

1 Introduction

This document may be considered to be a somewhat informal (no pun intended) appendix with José Faria's "Formal Methods Report" [8] (which is mostly self-contained and should be consulted whenever implicit references are encountered). The former report gives an overview of the formal verification of the Harris implementation of non-blocking linked lists [9].

In the case study, both TLA/TLC and Promela/Spin are used to model and verify the concurrent algorithm. Both formalisms were developed specifically for the formal verification of concurrent systems and share the main paradigm of model checking. The comparison criteria used in the report include productivity, tool performance and expressiveness of the language, among a handful of others.

In our report, we give an overview of how we developed a formal specification of the same algorithm using the B method [1]. While a model checker for B also exists, the B method is based on the classical paradigms of weakest preconditions [7] and refinement [5]. A rather obvious drawback of B is that there is no direct support for concurrency. However, it turns out that very few changes to the language and the tool suffice to allow for the specification and verification of concurrent systems [2, 3, 4].

The resulting extension of B is sometimes called Event B, but we will use both terms interchangeably. Until very recently, the (free) tool support for Event B was less than satisfactory, but in late May 2005 a new version of B4Free with concurrency support was released¹ and it is this version that we have used in our experiment.

Even though we have tried to obtain a B model which is to some extent unbiased by the TLA and Promela models (in order to fully exploit the

¹<http://www.b4free.com/>

possibilities of the formalism), we have heavily borrowed from ideas that were developed during the earlier experiments. We would also like to point out explicitly that our main goal was *not* to fully verify the correctness of the algorithm used in the case study, but rather to get a general feel for the limitations and capabilities of (Event) B for the specification and verification of concurrent systems. More specifically we have made certain restrictions and simplifications (not unlike the ones mentioned in [8]) and (due to time restrictions) we did not carry through the refinement to the final level of implementation.

Section 2 briefly outlines some of the key concepts of the B method. The following sections show how an abstract model of the list operations can be consecutively refined to develop a more concrete specification. Section 9 gives some conclusions and general thoughts on the use of Event B for the verification of concurrent software.

2 The B Method

An excellent comprehensive introduction to the B method is given in [6], while the B Book [1] is still the ultimate reference on the subject. Here we merely mention some of the major key concepts. Additional concepts will be introduced when needed.

The starting point of philosophy for the B method is that correct software should be accompanied by a mathematical proof of its correctness. The mathematical proof ensures that deadlock is avoided and that every event (or state transition) preserves the invariant predicate, which expresses the static data properties at the most abstract level. The weakest precondition calculus is used to generate a number of proof obligations from the invariant and the shape of the events. Many proofs can be discharged by the automatic theorem prover. The remaining verification conditions must be proved manually by the user.

In order to manage the complexity of both specification and verification, B models can be developed incrementally through refinement. Every refinement establishes a link between an abstract and a more concrete model (the *glueing invariant*). The refinement must preserve the abstract invariant but may also introduce additional concrete invariants. Keeping every separate refinement step small reduces the complexity of the proofs. An implementation is a special kind of refinement which can be translated to executable (C) code. It is usually the final and most concrete specification in the chain of stepwise refinements at varying levels of abstraction.

The logic of B is based on set theory with the axiom of choice. \mathbb{B} , \mathbb{N} and \mathbb{Z} are predefined sets and the corresponding properties are available in the theorem prover. Relations and functions are defined in terms of sets. Concepts like injections, surjections and inverse functions are all predefined. The invariant and other properties are expressed in first order logic with equality. The result is that little more than high school mathematics is required for all data modeling in B.

Every possible computation (i.e., a transition over state variables) corresponds to an observed event. Events are modeled by so-called general-

ized substitutions, whose semantics are defined in the weakest precondition calculus. Intuitively, every event has a guard which will trigger the event. When triggered most events will execute a (parallel composition of) assignment(s). An event can have no input or output parameters and every event is considered to be atomic. When a refinement introduces new concrete events, they are considered to refine a skip event in the abstract model.

The B4Free distribution basically consists of two powerful tools. The *batch* tool generates the proof conditions for a given B project, based on weakest precondition and refinement calculus. The *krt* tool is an automated theorem prover for first order logic. Note that there is a strict distinction between the logic used to generate the proof obligations (i.e., weakest precondition and refinement calculus) and the logic of the proof obligations themselves (i.e., pure first order logic). This implies that, at least in principle, the proof obligations generated by B could also be discharged by a third party theorem prover and the B theorem prover could also be used to discharge proof obligations generated by different systems using different logics (e.g., temporal logic, CSP, etc.).

B4Free can be used in combination with an XEmacs interface which supports managing B projects and allows to replay proofs when small changes are made to a model, among many other useful functions. There is also an interactive prover available with a very effective user interface called Click'n'Proof. Overall we were quite impressed by the maturity of the B4Free tools. There are little or no bells and whistles but the tools allow to get the job done in the most effective way.

3 Inserting and Deleting Keys

The concurrent algorithm we want to model consists of different processes accessing a common data structure in memory, i.e., a linked list. The two operations the linked list supports are the insertion of nodes to the list and the deletion of nodes from the list. The list is ordered by the keys of the nodes and no duplicate keys occur in the list. In a linked list, every node contains a key and a pointer to the next node, and probably also some other content which is irrelevant in this context.

If we abstract away from the memory and the order in the list, a list could be considered to be nothing more than a subset of the set of possible keys. This *data model* is very simple and abstract, and it corresponds with an equally simple *operational model*: inserting a node means adding a key to the subset and deleting a node corresponds to removing a key from the subset.

The set of all possible keys can be any totally ordered set, but we may as well assume that the keys are drawn from the set of natural numbers. This has the advantage that many useful properties of the natural numbers (together with properties of $<$, \leq , $>$, \geq , min and max) are readily available in both the automated and the interactive theorem prover.

Figure 1 shows the initial B model. This model should be fairly self-explanatory to any reader vaguely familiar with set-theory. The **ANY**-construct is used to express the guard of the events and the invariant

expresses a global safety property of the system. Two generated proof obligations ensure that the initialization establishes the invariant and that both events preserve the invariant. Both obligations are discharged by the automated theorem prover without any human interaction.

```

MODEL
  Keys
VARIABLES
  keys
INVARIANT
   $keys \subseteq \mathbb{N}$ 
INITIALISATION
   $keys := \{\}$ 
EVENTS
  delete =
    ANY  $e$  WHERE  $e \in keys$  THEN
       $keys := keys - \{e\}$ 
    END;
  insert =
    ANY  $e$  WHERE  $e \in \mathbb{N} - keys$  THEN
       $keys := keys \cup \{e\}$ 
    END
END

```

Figure 1: The initial model

In this report, we will restrict ourselves to (global) *safety* properties which guarantee the consistency of the linked list data structure. Note that for instance our initial model does not express the property that new keys can always be inserted to the list (which would require a (trivial) proof of the infinite cardinality of \mathbb{N}). Such (local) *liveness* properties are hard to express with a global invariant. We will come back to this restriction in later sections.

4 The Memory Model

The concept of a linked list is inherently connected to the concept of a memory model. From an abstract point of view, the memory consists of a set of memory locations (or addresses). The nodes of the list form a subset of those memory locations. There are two (different) special nodes, viz. the *head* and the *tail*, which initially belong to the list and can not be deleted. All other nodes (which are called normal nodes) contain a key.

This can be modeled by a partial injection *key* mapping nodes to the set of key values. The domain of *key* is the set of all normal nodes and the range corresponds to the subset *keys* of key values which was introduced in our initial model.

The preceding informal description of our memory model is formalized in the additional (concrete) invariant of Figure 2. Note that the set *locs* of memory locations must satisfy the property that it contains at least

two elements. This ensures that the initialization can establish the invariant, which demands that the *head* and *tail* nodes correspond to different memory locations.

```

REFINEMENT
  Memory
REFINES
  Keys
SETS
  locs
PROPERTIES
   $locs \neq \{\} \wedge \forall(x \bullet x \in locs \implies locs \neq \{x\})$ 
VARIABLES
  keys, nodes, hd, tl, key
INVARIANT
   $nodes \subseteq locs \wedge$ 
   $hd \in nodes \wedge$ 
   $tl \in nodes \wedge$ 
   $hd \neq tl \wedge$ 
   $key \in nodes \gg \mathbb{N} \wedge$ 
   $dom(key) = nodes - \{hd, tl\} \wedge$ 
   $ran(key) = keys$ 
INITIALISATION
  ANY  $x$  WHERE  $x \in locs$  THEN
    ANY  $y$  WHERE  $y \in locs - \{x\}$  THEN
       $hd := x$  ||
       $tl := y$  ||
       $nodes := \{x, y\}$  ||
       $key := \{\}$  ||
       $keys := \{\}$ 
    END
  END
EVENTS
  delete =
    ANY  $k$  WHERE  $k \in ran(key)$  THEN
      ANY  $n$  WHERE  $n \in nodes - \{hd, tl\} \wedge key(n) = k$  THEN
         $nodes := nodes - \{n\}$  ||
         $key := key - \{n \mapsto k\}$  ||
         $keys := keys - \{k\}$ 
      END
    END;
  insert =
    ANY  $k$  WHERE  $k \in \mathbb{N} - ran(key)$  THEN
      ANY  $n$  WHERE  $n \in locs - nodes$  THEN
         $nodes := nodes \cup \{n\}$  ||
         $key := key \cup \{n \mapsto k\}$  ||
         $keys := keys \cup \{k\}$ 
      END
    END
END

```

Figure 2: Introducing the memory model

The last line of the invariant is a so-called glueing invariant, which

establishes the link between the abstract and the concrete model. In this particular case a set (*keys*) is refined as the range of a function (*key*). While the set *keys* is needed in this concrete model in order to establish the link, it will not occur in any further refined models.

Note that the initial guards for the events are now strengthened with a condition for the nodes. In the case of an *insert*, any fresh location will do, but in the case of a *delete*, there is an obvious connection between the key and the node to be removed, viz. $key(n) = k$. In fact, since *key* is injective, we could do without the extra guard and just use $key^{-1}(k)$ for *n*, but our formulation exploits the symmetry between both events and is (hence) more readable, which will become more obvious when the model is further refined.

Among the 22 proof obligations generated by the system, 7 proofs could not be discharged automatically and required an (in this case painless and straightforward) interaction from the user.

5 Adding the Links to the List

In a (singly) linked list, a node contains more than just a key. Conceptually, the nodes form a chain where every node but the last points to the next node in the chain. This can be modeled with a relation between nodes, more specifically a partial injective function over nodes whose domain consists of all nodes minus the tail. The range of this *next* function consists of all nodes minus the head.

The concrete invariant of Figure 3 formalizes these ideas. We also introduce the main safety property of the system, i.e., the list should always stay consistent. More specifically, the *next* function must agree with the natural order of the keys. This is expressed with two new conditions in the invariant. Informally, the first condition states that the key of a given node must be less than the key of the next node. The second condition states that there can be no node whose key is in between the key of a given node and the key of the next node.

The assertion clause contains (non-inductive) properties which follow directly from the invariant. This allows to introduce *lemmas* which (once proved) can be used in different proofs. While it can be shown that the last condition of the invariant follows from the rest of the invariant, it is hard to imagine a proof that does not involve induction over the length of the list. Hence we include this condition in the invariant, which is inherently inductive in nature.

The initialization is the same as before but also initializes the *next* function as a single link pointing from the head to the tail node. Similarly, both the *delete* and the *insert* event now include an update of the *next* function. In the *delete* event, a guard is added to select the node *l* that points to the node to be removed. Updating the pointers is then a straightforward affair. The *insert* event is again similar to the *delete*, but selecting the *l*-node is more complicated because the node to be inserted is not yet present in the list and hence the keys of the nodes have to be compared.

To prove the correctness of this refinement, 24 proof obligations had

```

REFINEMENT
  LinkedList
REFINES
  Memory
VARIABLES
  nodes, hd, tl, key, next
INVARIANT
   $next \in nodes \gg nodes \wedge$ 
   $dom(next) = nodes - \{tl\} \wedge ran(next) = nodes - \{hd\} \wedge$ 
   $\forall (n \bullet n \in nodes - \{hd, tl\} \wedge next(n) \neq tl \implies (key(n) < key(next(n)))) \wedge$ 
   $\forall (m, n \bullet n \in nodes - \{hd, tl\} \wedge next(n) \neq tl \wedge$ 
     $m \in nodes - \{hd, tl\} \wedge key(n) < key(m) \implies$ 
     $(key(next(n)) \leq key(m))$ 
ASSERTIONS
   $\forall (n \bullet n \in nodes - \{tl\} \implies next(n) \neq n) \wedge$ 
   $(next(hd) \neq tl \implies key(next(hd)) = \min(ran(key))) \wedge$ 
   $(next^{-1}(tl) \neq hd \implies key(next^{-1}(tl)) = \max(ran(key))) \wedge$ 
   $(nodes - \{hd, tl\} \neq \{\} \implies next(hd) \neq tl)$ 
INITIALISATION
  ANY  $x$  WHERE  $x \in locs$  THEN
    ANY  $y$  WHERE  $y \in locs \wedge y \neq x$  THEN
       $hd := x \parallel tl := y \parallel$ 
       $nodes := \{x, y\} \parallel key := \{\} \parallel$ 
       $next := \{x \mapsto y\}$ 
    END
  END
EVENTS
  delete =
    ANY  $k$  WHERE  $k \in ran(key)$  THEN
      ANY  $n$  WHERE  $n \in nodes - \{hd, tl\} \wedge key(n) = k$  THEN
        ANY  $l$  WHERE  $l \in nodes - \{tl\} \wedge next(l) = n$  THEN
           $nodes := nodes - \{n\} \parallel$ 
           $key := key - \{n \mapsto k\} \parallel$ 
           $next := next - \{l \mapsto n\} - \{n \mapsto next(n)\}$ 
           $\cup \{l \mapsto next(n)\}$ 
        END
      END
    END
  insert =
    ANY  $k$  WHERE  $k \in \mathbb{N} - ran(key)$  THEN
      ANY  $n$  WHERE  $n \in locs - nodes$  THEN
        ANY  $l$  WHERE  $l \in nodes - \{tl\} \wedge$ 
           $(l \neq hd \implies key(l) < k) \wedge$ 
           $(next(l) \neq tl \implies k < key(next(l)))$  THEN
           $nodes := nodes \cup \{n\} \parallel$ 
           $key := key \cup \{n \mapsto k\} \parallel$ 
           $next := (next - \{l \mapsto next(l)\})$ 
           $\cup \{l \mapsto n\} \cup \{n \mapsto next(l)\}$ 
        END
      END
    END
END

```

Figure 3: Adding the links

to be proved, among which 18 needed interaction. While the required interaction is in some cases limited to clicking a single button, there were also some rather hard proofs, often involving many case distinctions to account for the special cases of the head and tail nodes.

6 Introducing Processes

Up until now, we have modeled the *delete* and *insert* events atomically, i.e., they are considered to happen within a single time instant. In such an abstract time model, it does not really matter whether there are several different processes accessing the list, since a finer time granularity is needed to detect the interference between the processes.

In the previous sections, we have shown how the data model and the operational model can be refined by introducing new state variables and extending the event bodies. In order to refine the time model, we need to introduce new events which refine a skip event in the abstract model. Along with the new events, we will introduce the concept of concurrently running processes which may be accessing the list.

Instead of completely replacing the abstract *delete* and *insert* events by concrete events of finer granularity, we will refine the abstract events in such a way that they become the final step in the algorithm. The new concrete events represent the preparation phase that every process has to go through before the final step can occur.

More specifically, a process must first announce its intentions, i.e., select a list operation and pick a goal key (phase 1). Once this initial event (*startdel* or *startins*) has occurred, a process may search for a left and a right node in the list (phase 2). Finally, when the *search* event has happened, a *delete* or *insert* event may occur (if the searched nodes still obey the conditions) (phase 3).

In order to impose the order of the events, we introduce the global state variables *del*, *ins* and *searched*, which are all subsets of the set of *processes*. After phase 1 a process will be in either *del* or *ins* and after phase 2 it will also be in *searched*. When phase 3 is finally completed, the process will return to the pool of inactive processes. The process guards ensure that the events can only occur for processes which have completed the required phase(s) and are hence in the right subset.

The goal key, the left node and the right node are specified as (total) functions over subsets of processes (resp. *gkey*, *lt* and *rt*). This approach allows to model local process variables as global state variables. The goal key is set during phase 1 (hence the domain of *gkey* is $del \cup ins$). Similarly, the left and right nodes are set during phase 2 and the domain of both *lt* and *rt* is the set *searched*.

After reading the above, the model of Figure 4 should again be self-explanatory. Note that the invariant expresses only two of the conditions which the left and right nodes must obey. The condition that the right node is the next node of the left node is not an invariant because this condition may be invalidated by events corresponding to different processes. The left and right node of a given process may even be deleted from the list by a different process, which is why the range of *lt* and *rt* refers to

REFINEMENT*Processes***REFINES***LinkedList***SETS***processes***VARIABLES***nodes, hd, tl, key, next, del, ins, searched, lt, rt, gkey***INVARIANT**

$$\begin{aligned}
& del \subseteq processes \wedge ins \subseteq processes \wedge \\
& searched \subseteq del \cup ins \wedge del \cap ins = \{\} \wedge \\
& lt \in searched \rightarrow locs - \{tl\} \wedge rt \in searched \rightarrow locs - \{hd\} \wedge \\
& gkey \in del \cup ins \rightarrow \mathbb{N} \wedge \\
& \forall (p \bullet p \in searched \implies (lt(p) \in \text{dom}(key) \implies key(lt(p)) < gkey(p))) \wedge \\
& \forall (p \bullet p \in searched \implies (rt(p) \in \text{dom}(key) \implies gkey(p) \leq key(rt(p))))
\end{aligned}$$
ASSERTIONS

$$\begin{aligned}
& \forall (m, n \bullet n \in nodes - \{tl\} \wedge m \in nodes - \{hd, tl\} \wedge (n \neq hd \implies key(n) < key(m)) \wedge \\
& \quad (next(n) \neq tl \implies key(m) \leq key(next(n))) \implies (m = next(n)))
\end{aligned}$$
INITIALISATION**ANY** *x* **WHERE** *x* \in *locs* **THEN****ANY** *y* **WHERE** *y* \in *locs* \wedge *y* \neq *x* **THEN***hd* := *x* || *tl* := *y* || *nodes* := {*x, y*} ||*next* := {*x* \mapsto *y*} || *key* := {} ||*del* := {} || *ins* := {} || *searched* := {} ||*lt* := {} || *rt* := {} || *gkey* := {}**END****END****EVENTS***startdel* =**ANY** *p* **WHERE** *p* \in *processes* - *del* - *ins* **THEN****ANY** *k* **WHERE** *k* \in \mathbb{N} - $\text{ran}(gkey)$ **THEN***del* := *del* \cup {*p*} || *gkey* := *gkey* \cup {*p* \mapsto *k*}**END****END;***startins* =**ANY** *p* **WHERE** *p* \in *processes* - *del* - *ins* **THEN****ANY** *k* **WHERE** *k* \in \mathbb{N} - $\text{ran}(gkey)$ **THEN***ins* := *ins* \cup {*p*} || *gkey* := *gkey* \cup {*p* \mapsto *k*}**END****END;***search* =**ANY** *p* **WHERE** *p* \in (*del* \cup *ins*) - *searched* **THEN****ANY** *r* **WHERE***r* \in *nodes* - {*hd*} \wedge (*r* \neq *tl* \implies *gkey*(*p*) \leq *key*(*r*))**THEN****ANY** *l* **WHERE***l* \in *nodes* - {*tl*} \wedge (*l* \neq *hd* \implies *key*(*l*) < *gkey*(*p*)) \wedge *next*(*l*) = *r***THEN***lt* := *lt* \cup {*p* \mapsto *l*} || *rt* := *rt* \cup {*p* \mapsto *r*} ||*searched* := *searched* \cup {*p*}**END****END****END;**

Figure 4: Introducing processes

locs rather than *nodes*.

Also note that we have added a concrete assertion. While this additional lemma actually follows from the invariant of the *abstract* model, it is most useful when proving the correctness of the *concrete* refinement.

```

delete =
ANY p WHERE p ∈ del ∩ searched THEN
  ANY k WHERE k ∈ ran(key) ∧ gkey(p) = k THEN
    ANY n WHERE n ∈ nodes - {hd,tl} ∧ key(n) = gkey(p) THEN
      ANY l WHERE l ∈ nodes - {tl} ∧ lt(p) = l ∧
        (l ≠ hd ⇒ key(l) < gkey(p)) THEN
          ANY r WHERE r ∈ nodes - {hd} ∧ rt(p) = r ∧
            (r ≠ tl ⇒ gkey(p) ≤ key(r)) ∧
            next(l) = r THEN
            nodes := nodes - {n} ||
            key := key - {n ↦ gkey(p)} ||
            next := next - {l ↦ n} - {n ↦ next(n)}
              ∪ {l ↦ next(n)} ||
            gkey := gkey - {p ↦ gkey(p)} ||
            searched := searched - {p} ||
            del := del - {p} ||
            lt := lt - {p ↦ lt(p)} ||
            rt := rt - {p ↦ rt(p)}
          END
        END
      END
    END
  END;
insert =
ANY p WHERE p ∈ ins ∩ searched THEN
  ANY k WHERE k ∈ ℕ - ran(key) ∧ gkey(p) = k THEN
    ANY n WHERE n ∈ locs - nodes - ran(lt) - ran(rt) THEN
      ANY l WHERE l ∈ nodes - {tl} ∧ lt(p) = l ∧
        (l ≠ hd ⇒ key(l) < gkey(p)) THEN
          ANY r WHERE r ∈ nodes - {hd} ∧ rt(p) = r ∧
            (r ≠ tl ⇒ gkey(p) ≤ key(r)) ∧
            next(l) = r THEN
            nodes := nodes ∪ {n} ||
            key := key ∪ {n ↦ gkey(p)} ||
            next := (next - {l ↦ next(l)})
              ∪ {l ↦ n} ∪ {n ↦ next(l)} ||
            gkey := gkey - {p ↦ gkey(p)} ||
            searched := searched - {p} ||
            ins := ins - {p} ||
            lt := lt - {p ↦ lt(p)} ||
            rt := rt - {p ↦ rt(p)}
          END
        END
      END
    END
  END
END

```

Figure 5: Introducing processes (ctd.)

The events of Figure 5 are quite similar to their abstract counterparts. The main difference is that there is a new guard for the processes and that there are additional assignments to reset the local process variables and to set the process status to inactive.

Furthermore, there are additional guards which ensure that the left and right nodes still obey the required conditions. Actually, this “check” makes the search event somehow obsolete, since the search is repeated implicitly in the guards. Obviously, these guards will have to be refined eventually if we want to obtain a specification that can be translated to executable code (with every event representing an atomic instruction). However, postponing the refinement of the guard until a later stage of development allows to reduce the complexity of the proofs at the current stage.

Our specification requires that the fresh location which is selected by the *insert* event is not within the range of the *lt* and *rt* functions. A location within the range of these functions is still being used by some other process, which could lead to serious problems (see the memory management drawback outlined in [8]).

We recall that liveness properties are not being considered in our specification. For instance, if a deleting process *a* completes the search phase and a process *b* deletes the left node of process *a* from the list, process *a* will never be able to complete the delete phase (since the left node no longer belongs to the set of nodes).

In fact, in our current model, certain processes will be stuck forever. In order to avoid this, we should add complimentary events. For instance, if the left node of a given process no longer belongs to the set of nodes, the process should search again. Such additional events will allow a process to eventually satisfy the guard of one of the events that are currently in our model.

While the complimentary events will ensure that every progress will be able to make progress, these events will certainly not invalidate the safety properties (the list is only modified in a final *delete* or *insert* event). Since we restrict ourselves to safety properties, we may as well not bother about the blocked processes and complimentary events.

No less than 51 proof obligations were generated for this refinement and 24 of those required interaction. However, most of these 24 remaining conditions could also be proved automatically by simply extending the time limit for the automatic theorem prover.

7 Further Refining the Insert Event

While the delete and insert operations are in a sense dual operations, the insert operation is easier to implement, since no marking is required and a single CAS-instruction suffices. In our next refinement, we will restrict our attention to the *insert* event.

The refinement of Figure 6 is almost identical to the previous one. The only difference is that the guards of the *insert* event are rewritten in a way that more strongly resembles the implementation. The guards for *k*, *l*

```

REFINEMENT
  Insert
REFINES
  Processes
VARIABLES
  nodes, hd, tl, key, next, del, ins, searched, lt, rt, gkey
INITIALISATION
  ANY x WHERE  $x \in locs$  THEN
    ANY y WHERE  $y \in locs \wedge y \neq x$  THEN
      hd := x ||
      tl := y ||
      nodes := {x, y} ||
      next := {x ↦ y} ||
      key := {} ||
      del := {} ||
      ins := {} ||
      searched := {} ||
      lt := {} ||
      rt := {} ||
      gkey := {}
    END
  END
EVENTS
  insert =
    ANY p WHERE  $p \in ins \cap searched$  THEN
      ANY n WHERE  $n \in locs - nodes - \text{ran}(lt) - \text{ran}(rt)$  THEN
        SELECT ( $rt(p) = tl \vee (rt(p) \in \text{dom}(key) \wedge key(rt(p)) \neq gkey(p))$ )  $\wedge$ 
          ( $lt(p) \in \text{dom}(next) \wedge next(lt(p)) = rt(p)$ ) THEN
          nodes := nodes  $\cup$  {n} ||
          key := key  $\cup$  {n ↦ gkey(p)} ||
          next := (next - {lt(p) ↦ rt(p)} )
             $\cup$  { lt(p) ↦ n }  $\cup$  { n ↦ rt(p) } ||
          gkey := gkey - {p ↦ gkey(p)} ||
          ins := ins - {p} ||
          searched := searched - {p} ||
          lt := lt - {p ↦ lt(p)} ||
          rt := rt - {p ↦ rt(p)}
        END
      END
    END
  END

```

Figure 6: Refining insert

and r from the abstract model are replaced by a single select-guard which checks the conditions for the left and right nodes.

Note that we are now quite close to an atomic implementation. Most of the updates in the event do not have to be implemented explicitly but instead happen implicitly, e.g., when the local process variables go out of scope at the end of the method. The only exception is the update of the *next* function, which essentially updates the pointer $lt(p) \mapsto rt(p)$ to $lt(p) \mapsto n$. This pointer update can be combined with the second line of the **SELECT**-construct into a single CAS-operation (see */*C2** in the implementation).

What remains to be done is to split off both the first line of the **SELECT**-construct (see */*T1** in the implementation) and the addition of the $n \mapsto rt(p)$ pointer to separate events (which will not modify the actual list).

In order to prove that we have obtained a correct refinement of our previous model, 7 conditions had to be proved. This low number is due to the fact that in this development step we restricted ourselves to a (rather small) refinement of a single event and we did not add any concrete invariants. However all 7 proof obligations needed some human interaction.

8 Marking the Nodes

Let us now take a closer look at the delete operation. This operation can not be implemented by a single CAS-instruction as in the case of an insert operation. Instead a separate marking (or logical deletion) phase is required before a node can be physically deleted.

In order to refine our specification with the concept of marking nodes, we add an event *mark*, which has to occur between every *search* and *delete* event of the same (deleting) process. This order of events can be imposed by adding a set of processes called *hasmarked*, which is always a subset of both *del* and *searched*. We also introduce a subset *marked* of locations which represents the set of logically deleted nodes.

The invariant of Figure 7 states that there can never be two (deleting) processes that mark the same node. The reasoning behind this is that the second process should simply physically delete the marked node instead of marking it again.

However, this is not what happens in our current model, since the guard of the *delete* event in Figure 8 prevents processes that have not marked a node to physically delete a node, even when it is marked by a different process. In fact, our model currently specifies a *blocking* algorithm. However, we already know from earlier refinements that, while processes can get stuck, this will never invalidate the safety properties that we want to verify.

Because of the blocking nature of our algorithm specification, the properties that are satisfied by a marked node can not be invalidated by another process and hence are also part of the invariant. More specifically, a process that has successfully completed its marking phase has a right node which is marked and whose key equals the goal key of the deleting process, i.e., the right node is the node to be deleted from the list.

REFINEMENT*Mark***REFINES***Processes***VARIABLES***nodes, hd, tl, key, next, del, ins, searched, lt, rt, gkey, hasmarked, marked***INVARIANT**

$$\begin{aligned}
& \text{hasmarked} \subseteq \text{del} \cap \text{ssearched} \wedge \text{marked} \subseteq \text{locs} \wedge \\
& \forall (p, q \bullet p \in \text{hasmarked} \wedge q \in \text{hasmarked} \wedge p \neq q \implies (\text{rt}(p) \neq \text{rt}(q))) \wedge \\
& \forall (p \bullet p \in \text{hasmarked} \implies (\text{rt}(p) \neq \text{tl} \wedge \\
& \quad \text{rt}(p) \in \text{dom}(\text{key}) \wedge \text{key}(\text{rt}(p)) = \text{gkey}(p) \wedge \text{rt}(p) \in \text{marked}))
\end{aligned}$$
INITIALISATION**ANY** x **WHERE** $x \in \text{locs}$ **THEN****ANY** y **WHERE** $y \in \text{locs} \wedge y \neq x$ **THEN** $hd := x \parallel tl := y \parallel \text{nodes} := \{x, y\} \parallel$ $\text{next} := \{x \mapsto y\} \parallel \text{key} := \{\} \parallel$ $\text{del} := \{\} \parallel \text{ins} := \{\} \parallel \text{ssearched} := \{\} \parallel$ $lt := \{\} \parallel \text{rt} := \{\} \parallel \text{gkey} := \{\} \parallel$ $\text{hasmarked} := \{\} \parallel \text{marked} := \{\}$ **END****END****EVENTS***search =***ANY** p **WHERE** $p \in (\text{del} \cup \text{ins}) - \text{ssearched}$ **THEN****ANY** r **WHERE** $r \in \text{nodes} - \{hd\} - \text{marked} \wedge$ $(r \neq tl \implies \text{gkey}(p) \leq \text{key}(r))$ **THEN****ANY** l **WHERE** $l \in \text{nodes} - \{tl\} - \text{marked} \wedge$ $(l \neq hd \implies \text{key}(l) < \text{gkey}(p)) \wedge$ $\text{next}(l) = r$ **THEN** $lt := lt \cup \{p \mapsto l\} \parallel$ $rt := rt \cup \{p \mapsto r\} \parallel$ $\text{ssearched} := \text{ssearched} \cup \{p\}$ **END****END****END;***mark =***ANY** p **WHERE** $p \in \text{del} \cap \text{ssearched}$ **THEN****ANY** n **WHERE** $n \in \text{locs} - \text{nodes} - \text{ran}(lt) - \text{ran}(rt)$ **THEN****SELECT** $\text{rt}(p) \neq tl \wedge \text{rt}(p) \in \text{dom}(\text{key}) \wedge \text{key}(\text{rt}(p)) = \text{gkey}(p) \wedge$ $\text{rt}(p) \notin \text{marked}$ **THEN** $\text{marked} := \text{marked} \cup \{\text{rt}(p)\} \parallel$ $\text{hasmarked} := \text{hasmarked} \cup \{p\}$ **END****END****END;**

Figure 7: Marking the nodes

```

delete =
ANY  $p$  WHERE  $p \in \text{hasmarked}$  THEN
  SELECT  $lt(p) \in \text{dom}(\text{next}) \wedge \text{next}(lt(p)) = rt(p)$  THEN
     $\text{nodes} := \text{nodes} - \{rt(p)\}$  ||
     $\text{key} := \text{key} - \{rt(p) \mapsto gkey(p)\}$  ||
     $\text{next} := \text{next} - \{lt(p) \mapsto rt(p)\} - \{rt(p) \mapsto \text{next}(rt(p))\}$ 
       $\cup \{lt(p) \mapsto \text{next}(rt(p))\}$  ||
     $gkey := gkey - \{p \mapsto gkey(p)\}$  ||
     $\text{hasmarked} := \text{hasmarked} - \{p\}$  ||
     $\text{searched} := \text{searched} - \{p\}$  ||
     $\text{del} := \text{del} - \{p\}$  ||
     $lt := lt - \{p \mapsto lt(p)\}$  ||
     $rt := rt - \{p \mapsto rt(p)\}$  ||
     $\text{marked} := \text{marked} - \{rt(p)\}$ 
  END
END;
insert =
ANY  $p$  WHERE  $p \in \text{ins} \cap \text{searched}$  THEN
  ANY  $n \in \text{locs} - \text{nodes} - \text{ran}(lt) - \text{ran}(rt)$  THEN
    SELECT
       $(rt(p) = tl \vee (rt(p) \in \text{dom}(\text{key}) \wedge \text{key}(rt(p)) \neq gkey(p))) \wedge$ 
       $(lt(p) \in \text{dom}(\text{next}) \wedge \text{next}(lt(p)) = rt(p))$  THEN
         $\text{nodes} := \text{nodes} \cup \{n\}$  ||
         $\text{key} := \text{key} \cup \{n \mapsto gkey(p)\}$  ||
         $\text{next} := (\text{next} - \{lt(p) \mapsto rt(p)\})$ 
           $\cup \{lt(p) \mapsto n\} \cup \{n \mapsto rt(p)\}$  ||
         $gkey := gkey - \{p \mapsto gkey(p)\}$  ||
         $\text{ins} := \text{ins} - \{p\}$  ||
         $\text{searched} := \text{searched} - \{p\}$  ||
         $lt := lt - \{p \mapsto lt(p)\}$  ||
         $rt := rt - \{p \mapsto rt(p)\}$ 
      END
    END
  END
END

```

Figure 8: Marking the nodes (ctd.)

The *mark* event corresponds to a successful iteration of the */B4*/* loop in the delete method. The instructions */T1*/* and */C3*/* can be clearly recognized in our specification, but the event should be further refined before it can be translated to executable code.

The *delete* event now corresponds to the final instruction */C4*/* of the delete method. Again the essential update of the event could be combined with the main guard to be finally implemented by a single CAS-operation, much like in the case of the *insert* event.

For this final refinement, only 10 of the 50 generated proof obligations needed some limited interaction to be proved.

9 To B Or Not To B?

While our final refinement resembles the general structure of the algorithm, obviously some work remains to be done in order to obtain a full specification. Ideally, the final step in a B development should be an implementation. Technically, an implementation looks just like a refinement but the instruction set of generalized substitutions is restricted in such a way that the events can be translated to executable code.

In order to arrive at that lowest level of abstraction, we should refine our events even further. For instance we did not refine the *search* event at all, because the search method does not modify the list and is hence rather uninteresting for the verification of safety properties.

While the idea of a refinement chain ending in an implementation certainly looks interesting in theory, we have not yet tested this feature due to time restrictions. In practice, we might encounter problems with expressing concepts like pointers and the CAS-instruction within the restricted B language.

Since the non-locking nature of an algorithm can only be expressed in a natural way by local liveness properties, our restriction to the verification of global safety properties is a serious one. While it may be possible to express liveness properties by using explicit counters in the invariant, this solution does not appear to be attractive.

In [4], the authors discuss the possible addition of a *variant* and *modalities* clause to the B language. The addition of these clauses would allow us to express liveness properties in the most natural way. Adding such liveness properties would then force us to add the complimentary events as outlined in section 6. However, these new clauses do not seem to be supported (yet) by the tools we have used.

The concept of refinement makes B very well suited for developing concrete algorithms and programs from an abstract specification. Working the other way around, i.e., developing an abstract specification and a refinement chain which ends in a given implementation, does not appear to be the most natural way to work. While this argument also holds for TLA (and to a lesser extent for PROMELA), the B method may be even less suitable for ad hoc verification.

A strong selling point for the B method is that the language is very simple. Every programmer should be familiar with set-theory and first-order logic and this is essentially all that is required to write specifications

in B. Some knowledge of weakest precondition calculus and refinement is certainly useful but not essential since all proof obligations are generated automatically (and many are proved automatically too).

We were able to get acquainted with the B language and tools, develop the refinement chain (including all proofs) and write this report in less than a month's time. Due to the experience gained in this experiment, developing a new algorithm of similar complexity should take considerably less effort and time. The fact that B is relatively easy to pick up and to use effectively is probably the main reason why the B method has been used successfully in many industrial projects.

Obviously symbolic model checking also allows to prove software correct and has the advantage that the method is fully automatic. There are however inherent limitations to such an approach. Most notably the state explosion problem may force the user to either lower the problem size or to increase the computing power. Depending on the context, both options may be undesirable at best and unacceptable at worst.

Within the B method, there are no assumptions whatsoever about the size of the system. The price to pay is that the proofs that can not be discharged by the automated theorem prover may be non-trivial and hence time consuming. On the other hand, manually proving software properties may bring subtleties to the surface and may provide insight which would be hard to obtain otherwise (much like in traditional mathematics).

References

- [1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.
- [2] J.-R. Abrial. Extending b without changing it (for developing distributed systems). *Proceedings of 1st Conference on the B method*, pages 169–191, 1996.
- [3] J.-R. Abrial. Event driven distributed program construction. MATISSE project, 2001.
- [4] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in b. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 83–128, London, UK, 1998. Springer-Verlag.
- [5] R.-J. Back. On correct refinement of programs. *J. Comput. Syst. Sci.*, 23(1):49–68, 1981.
- [6] D. Cansell and D. Méry. Foundations of the b method. *Computing and Informatics*, 22(3–4):221–256, 2003.
- [7] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
- [8] J. S. Faria. Formal methods report. Internal report, Ghent University, June 2005.
- [9] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–314, 2001.