B vs. Coq to prove a Garbage Collector

L. Burdy

GEMPLUS
Avenue du Pic de Bertagne - 13881 Gémenos Cedex - France
lilian.burdy@gemplus.com

Abstract. This paper presents a comparison between two formal methods : the B method and the Coq proof assistant on their usability in an industrial context. Those methods have been used to formalise a garbage collector algorithm and prove its safety property. They are compared on two aspects : formalisation and proof. At each step, the two models are compared in terms of readability, easiness of proof and maintainability. Metrics on both development are presented.

1 Introduction

The use of formal method is a growing activity in the industrial world. Typically in the smart card domain, the requisite of using formal methods to obtain high level certificate has obliged industries to study and evaluate different formal methods. The criteria are both efficiency of formalisation and ease with which it could be learned and used by engineers and evaluators. Currently those methods are commonly used by researcher, but the aim is to transfer this technology to engineers, still they have to use them in their current activity. For instance, formal methods have to be used in a development process to obtain an EAL5 certificate in the Common Criteria. Motré and Teri present such a work in [MT00].

Currently B and Coq, but also PVS are studied in the Gemplus research lab. In this study, the comparison is only pragmatic until the aim is to make formal modelling a current activity of an engineer in an industrial process.

This paper presents a pre-study concerning the formalisation of a garbage collector (GC) algorithm with the proof of one safety property : that is the memory cells which are collected by the algorithm are not accessible from a root of the memory.

Different GC have already been modelled with different formal techniques : in [HS97], Havelund and Shankar have modelled and proved the safety property of a mark and sweep with PVS, in [Jac98] Jackson has modelled a tricolour mark and sweep and proved the safety and liveness properties with PVS and in [MD99] Moreau and Duprat have modelled a distributed reference counting algorithm and proved the safety and liveness properties with Coq.

The distinctive features of this paper are :

- A parallel presentation of two formal models using usually opposed formal methods ; and
- The will to compare those methods from an industrial point of view : that is in a way where the difficulty of using does not overflow the contribution of formal methods.

We present the algorithm we have chosen to formalise, then its formalisation in Coq and in B with a comparison of the two formalisations, then we present the proof of the safety property and how it has been performed in Coq and in B. The choice to distinguish the specification from the proof is only to clarify the presentation, in fact those two activities have been performed side by side.

To put into perspective the results presented here, I present my background with Coq and B. With Coq, I have had the theoretical background bases and a month training on the proof assistant. With B, I had four years practice in an industrial context and a PhD ([Bur00]). Another point is that the Coq model has been developed before the B one. One can consider that this background will lead to a vote through for the B method by plebiscite...

2 A Mark and Sweep Algorithm

We have chosen to prove a mark and sweep algorithm. This algorithm is described in [HS97]. Its main advantage is its simplicity and its main drawback is to be inefficient since the mark phase is very costly. We describe it briefly beginning with the memory representation, then the mutator which adds new links in the memory and finally the collector which frees non accessible nodes.

The Memory : The memory is represented as a set of nodes. For our purpose, a node is simply a list of pointer to other nodes (see figure 1), we do not care about the size of the node neither about the information it contains. We are only interested in the graph of pointers between nodes. Some nodes are considered as roots, those nodes have not to be pointed to be considered as accessible. We attach other information to nodes, their freeness and also, needed by the mark and sweep algorithm their colour (black or white). In the remainder of this paper, we say that a node is accessible if it is a root or if there exists a path between this node and a root.

The Mutator : The mutator represents the program or the system activity : it adds a new link between two nodes if the target is an accessible node. Then it colours this target in black.

The Collector : The collector is a mark and sweep algorithm. It is defined in three steps.

1. Colour each root black.

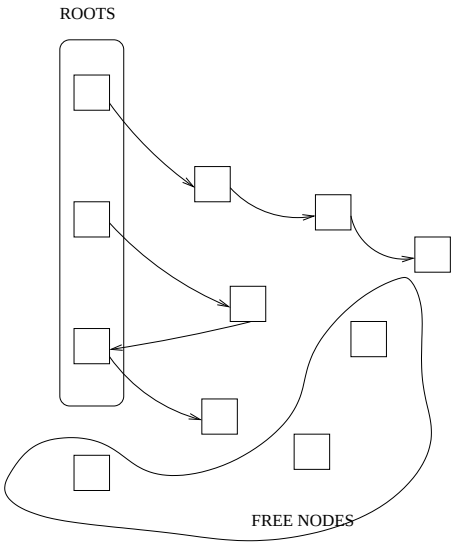

Fig. 1. Memory represented as a graph

2. Examine each pointer in succession. If the source is black and the target is white, colour the target black until no target have been coloured.
3. Examine each node in succession. If a node is white, make it free ; if it is black, colour it white.

The marking phase consists on the two first steps, the sweeping phase is the last step. The algorithm is really not efficient since the memory has to be scanned many times to mark the nodes (step 2).

3 Formalisation

The formalisation is presented in three parts : the memory description which is the static part and the two process descriptions : the mutator and the collector which form the dynamic part. For each part, we present the Coq formalisation, the B model and the results of the comparison.

To present the B models, we use the mathematical notation since the ASCII one is not clearly understandable by non-expert. Furthermore, we explain the non classical set theory operator used in the formalisation.

3.1 Memory Description

The description of the memory is the static part of the formalisation. The memory representation that we formalise is the memory informally described in the previous part.

We present the Coq formalisation, using inductive set declaration :

Inductive colour : Set :=
white : colour | black : colour.
Inductive l_cell : Set :=
null_c : l_cell | pointer : nat → l_cell → l_cell.
Inductive node : Set :=
cons_n : (id : nat) (co : colour) (free : bool) (root : bool) (list : l_cell) node.
Inductive memory : Set :=
null_m : memory | cons_m : node → memory → memory.

memory is the type of memories. A memory is defined as a list of nodes ended by the *null_m* constructor. A node contains an identifier, a colour, a first boolean indicating if it is free, a second boolean indicating if it is a root and a list of cell. A cell contains a pointer to a node identifier (a natural number).

We present the B formalisation, using variables to represent the memory :

SETS
memory;
$t\_colour = \{c\_black, c\_white\}$

ABSTRACT_VARIABLES
colour, free, nodes, pointer, roots

INVARIANT
$nodes \subset memory$ &
$colour \in nodes \rightarrow t\_colour$ &
$free \subset nodes$ &
$roots \subset nodes$ &
$pointer \in nodes \leftrightarrow nodes$

The memory is declared as a *SET* ; that is the way, in B, to have a finite and non empty set of objects which can be used as a type. The nodes are represented by a subset of memory objects. The attributes of nodes are described with variables : the colour is given by a total function from nodes to an enumerated set of colours, the free nodes and the roots are described by subsets of the nodes. The pointers between nodes are described by a relation.

The Coq description of the memory only describes the memory architecture, it does not describe the static properties of the memory. For example, this formalisation allows to have a memory with nodes with the same identifier. To avoid this inconsistency, we have added a property called *well_founded* which prevent this kind of memory and we have proved that this property is an invariant of our system. In B, this property is given by construction (a set cannot contain two instances of the same element). Another example : the Coq formalisation allows to have a memory with nodes with pointers out of the memory. A priori, this fact is not relevant since we did not have to add an invariant to suppress this kind of memory. Nevertheless, in B the *INVARIANT* clause contains the types of the

variables and also their properties. In the present case, the *pointer* variable is defined as a relation between nodes and not between nodes and memory.

In the Coq formalisation, we have a type for the memories in a general way. In B, the variables represent the state of one memory. The quantification is implicit in B ; in Coq all the theorems begin with : "for all memory m ...". Moreover, the *nodes* variable represents the set of the nodes of the memory in the current state of the B machine, in Coq *node* represents how a node is structured.

Thus, we can consider that the Coq formalisation is closer to the informal description presented in the previous chapter where the memory is presented in this way. Even so, the two specifications are not very different. One can say that the Coq approach is closer to the informal specification : the memory is described in a structural way, the specification contains an entry point : *memory*. The variables of the B machine are not presented as connected together. All of them are needed to represent the state of the memory. As an anecdote, one can notice that two models are in reverse order : from the pointers to the memory in Coq and from the memory to the pointers in B.

3.2 The Mutator Description

The mutator is the process which adds a new link between two nodes. In Coq, the mutator is formalised by an inductive definition with one constructor :

```
Inductive mutator : memory → memory → Prop :=
  alloc : (m,m',m" : memory; n,k : nat)
          (is_in n m) →
          (is_accessible k m) →
          (assign m n k m') →
          (colour black m' k m") →
          (mutator m m").
```

The mutator describes a property between two memories, one can interpret the second one as the result obtained after the mutator has achieved on the first. m represents the initial memory, m' an intermediate state and $m"$ the result of the mutation. This definition means : to obtain $m"$ from m, take any natural number n which is in m, take any natural number k which is accessible in m, then m' is m with a link between n and k. Finally colour the node k of m' in black and this will give you $m"$.

is_in, *is_accessible*, *assign* and *colour* are all inductive declarations. We describe them briefly :

- *(is_in n m)* holds if n is in the identifier list of the memory m.
- *(is_accessible k m)* holds if there is a path composed of pointers from a root to the node k in the memory m.
- *(assign m n k m')* holds if m' is the memory m with a new link between the nodes n and k.
- *(colour black m k m')* holds if m' is the memory m where the node k is coloured in black.

We do not present them completely here, only the *assign* declaration is given below.

In B, the mutator is formalised by an operation :

```
OPERATIONS
  mutator(n, k) =
  PRE
    n : nodes &
    k : memory &
    k : (closure(pointer))[roots]
  THEN
    pointer := pointer ∪ {(n, k)} ||
    colour(k) := c_black
  END
```

The operation has two parameters : a node n and a natural number k which is accessible from the roots. In the body of the operation, a link between n and k is added and the node k is coloured in black.

The accessibility of k is defined by a property *k : (closure(pointer))[roots]*. *closure(pointer)* is the set which contains the transitive and reflexive closure of the *pointer* relation. *(closure(pointer))[roots]* is the set which contains the image of the *roots* set under the *closure(pointer)* relation. So k is accessible if it belongs to the image of a node under the transitive and reflexive closure of the *pointer* relation.

The two formalisations are quite different. In Coq, we have to describe completely, at this level how to perform the accessibility of a node, whereas in B it is described by an expression (the definition of *is_accessible* in Coq is not simple neither really abstract). This remark can be applied to the four inductive definitions needed to model the mutator. For example, we give the formalisation of the assignment in Coq, this expression gives the complete specification of the assignment :

```
Inductive assign : memory → nat → nat → memory → Prop :=
  assign_ok_ok : (i,j : nat)
    (assign null_m i j null_m) |

  assign_ok : (i,j : nat; c : colour; l,r : bool; m : memory; n : l_cell)
    (assign (cons_m (cons_n i c l r n) m)
            i
            j
            (cons_m (cons_n i c l r (pointer j n)) m)) |

  assign_rec : (i,j,k : nat; m,m' : memory; n : l_cell; c :colour; l,r : bool)
    ¬k=i →
    (assign m i j m') →
    (assign (cons_m (cons_n k c l r n) m)
            i
```

j
(cons_m (cons_n k c l r n) m')).

To perform the assignment, the memory is checked looking for the identifier *i*, this is performed by *assign_rec*. When the head of the list contains the desired identifier, this node is changed and the pointer is added to the list. In B, we just have to tell that a couple is added to the *pointer* relation. Later in the formalisation, in the refinements, there will be a loop performing this research and this update. The abstraction (with the set theory) used in B is a well-suited feature to precise briefly what the function does. The Coq formalisation is non-ambiguous too but one can consider that it is more difficult for a reader to understand what the function really does. In B, we only have to prove the safety property at this abstract level. Then we will prove that the operation is correctly refined, which will ensure that the refinement will conserve the property. In Coq the proof will be performed at this unique level where all the details are given.

Another point : in Coq we have to introduce an intermediate state between the mutation and the colouring, in B we can describe the colouring in parallel with the assignment. On another part, one can consider that the mutator used in the Coq formalisation can be read at the same level than the B operation since one does not have to check how *is_accessible* is really described. Also giving the description of the assignment here can be compared to give the full implementation of the loop which adds the new link in B.

3.3 The Collector Description

The collector is the process which performs the mark and sweep algorithm. To prove that the algorithm is safe even if the mutator is activated during the collection, we have introduced the notion of action in the Coq formalisation :

```
Inductive action : steps → memory → memory → Prop :=
  mut: (m,m',m" : memory; n : steps)
       (action n m m') →
       (mutator m' m") →
       (action n m m") |

  step_1 : (m,m',m" : memory)
       (action st_0 m m') →
       (colour_root_black m' m") →
       (action st_1 m m") |

  step_2a : (m,m',m" : memory; n,p : nat)
       (action st_1 m m') →
       (colour_target m' m" p) →
       ¬p=O →
       (action st_1 m m") |

  step_2b : (m,m',m" : memory; n,p : nat)
```

(action st_1 m m') →
(colour_target m' m" O) →
(action st_2 m m") |

step_3 : (m,m',m" : memory)
(action st_2 m m') →
(sweep m' m") →
(action st_0 m m").

This *action* defines in a non-deterministic way events which act on the memory. Those events are :

- a mutation as described previously.
- a collection of non accessible nodes, this collection is performed in three steps :
  1. the colouring of the roots in black.
  2. the colouring in black of all the nodes pointed by a black node until all black nodes point black nodes, the third argument of *colour_target* is the number of coloured nodes (step_2a and step_2b).
  3. the sweep of all white nodes.

The mutator can occur at every moment, the collector is a sequence of three tasks that are performed sequentially : this is ensured by the step parameter.

The B specification contains two operations, one for the marking phase and the other for the sweeping :

OPERATIONS
mark =
BEGIN
colour :(colour : nodes → t_colour &
closure(pointer)[roots] ∩ colour^{-1}[{c_white}] = ∅) ||
step := c_step_1
END;

sweep =
PRE
step = c_step_1
THEN
free := free ∪ colour^{-1}[{c_white}] ||
colour := nodes{c_white} ||*
step := c_step_0
END

The sequencing of the two operations is assumed by the *step* variable. The specification of the mark operation is : "the colour variable becomes such that the accessible nodes are not white". The specification of the sweep operation is : "we add to the free nodes the white ones and we colour all the nodes in white".

The main difference between the two specifications consists on the mark algorithm description. In Coq we have described the algorithm where in B we have only described its result : all accessible nodes should be coloured in black. The fact that the mutator can occur during the collection is not clearly represented in B, still the machine contains now three operations, so the mutator can occur between mark and sweep operations. But in Coq one have proved that the mutator can also occur during the marking subtasks (for example during the root colouring). To obtain this detail of formalisation in B, one has to prove the safety property at each level in the refinement process but also to make the hypothesis that since the process that colour roots in black does not change neither the pointers nor the free nodes it cannot act on the safety of the memory.

Here we have only presented the first machine of the B specification, but the model contains six levels of refinement, so the reader, if he wants to check the complete algorithm has to read all the refinements. The refinement representation is a guide to read a B model : from the head machine to the last implementation. One can have different level of reading of a B model, if we only want to see that this GC algorithm is safe, we only have to look at the first machine but if we want to see if it is the algorithm we want that is modelled, one has to read the complete model. In Coq, the reader is not helped by the architecture ; on the other side, he gets all the information in the same file.

3.4 The property

We have focused on the safety property of the algorithm, that is all garbage nodes are not accessible from a root. In Coq we have proved the following theorem :

Theorem safety : (m,m' : memory; n : steps)
(action n m m') →
(well_founded m) →
(root_not_free m) →
(memory_safe m) →
(memory_safe m').

This theorem means that if an action is performed in any state *n*, and if the memory was safe before, then the new memory *m'* is still safe. To prove it we have added that the initial memory should be well founded and that in this memory the roots are not free. We have proved that those two properties were also invariant. To do that, we have the two following theorems :

Theorem well_founded : (m,m' : memory; n : steps)
(action n m m') →
(well_founded m) →
(well_founded m').

Theorem root_not_free_th : (m,m' : memory; n : steps)
(action n m m') →
(root_not_free m) →
(root_not_free m').

In B, we have added a new formula in the *INVARIANT* clause of our machine :

INVARIANT
$(closure(pointer))[roots] \cap free = \emptyset$

We can notice that this formula contains the two Coq properties : *memory_safe* and *root_not_free* since the identity relation is contained in the closure. We have already seen that the *well_founded* property is obtained by construction in B model.

4 Proof

The proof part is the most costly of the specification activities. It cannot clearly be separated from the writing of the specification, but nevertheless it is the proof activity where most of the time is spent.

The proofs in Coq were very long to write with many lines and many lemmas. The proof is really a "hand-writing" activity. The most activity in Coq is to find the invariants and to find the theorems to prove. To prove such theorems, many sublemmas have to be proved for each case even the obvious one. In B, if the invariant is not strong enough, the machine is modified and then the proof obligations are produced and the obvious ones are automatically proved.

The main problem with the B method and the Atelier B CASE tool is that you cannot completely perform some proofs. In some case, the rule base of the theorem prover does not contain the appropriate rules, and one has to write what is called a proof rule in the B community, which can also be called a lemma. This rule is put in a separate file and the proof is completed using this lemma. Then another tool is used to prove this lemma but this tool does not contain an interactive mode. So if some lemmas remain not proved by the tool, it remains you to prove them informally!

To conclude, one can say that we have had no real difficulty to perform the proof in Coq as long as the good invariants are founded and lemmas are defined at the adequate level. The main difficulty with the proof in B consists also on finding the loop invariants. In this development, there was a technical difficulty to use the *closure* operator since there was no pre-existing proof rules, so the proof have not been fully performed. It was quite a frustration to prove this B model. This problem is well-known and can really be an issue in a complete safety development.

5 Architecture and Metrics

The architecture of the Coq development is less structured than the B development with its refinement construction. To structure the Coq development, we use the *Require* command. The figure 5 presents the *Require* graph of this development. The B development is structured in refinement and implementation :

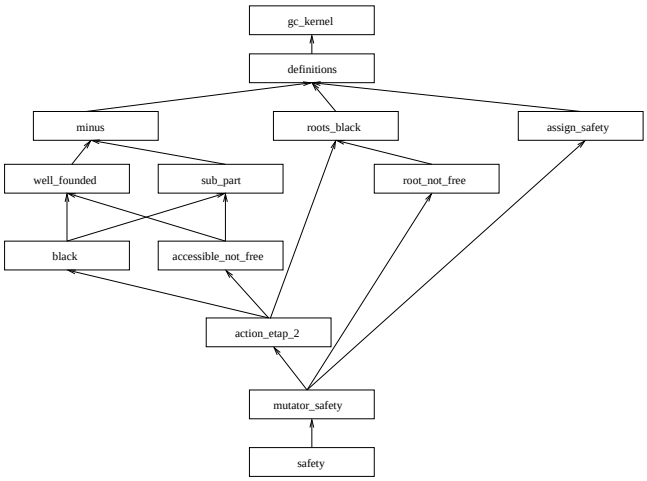

Fig. 2. The Coq vernaculars dependence graph

we have got six levels of machine with a level containing a refinement step. The intermediate implementation are introduced to simplify the proof obligations. The figure 5 presents the import graph of the development.

In table below, we present metrics about the two formalisations.

	Number of lines	Lines of specification	Lines of proof	Number of lemmas	Working days
Coq	7204	524	6680	229	16
B	1512	1200	312	353	6

In fact, in the 1200 specification lines of the B specification, only 383 are unique lines. This is due to the fact that one has to perform a lot of "copy and paste" when one use the refinement mechanism, mainly in order to simplify the proof obligations. By taking into account this fact, one can consider that the two specifications are quite equivalent in term of length.

The proof lines numbers can not be compared since the Coq proof assistant is not automated and not using tactics leads to very long proofs. To complete these metrics concerning the proof : during the B proof, 15 proof rules were added and 9 were proved by an off-prover tool. And at the end of the development, when no unproved lemmas remain, the tool automatically prove 80% of the proof obligations. This number is not really significant since the proof obligation number is tool-dependent.

6 Conclusion

First of all, no real blocking problem have been met in the development of these two formalisations. Even with no real background in using the Coq proof

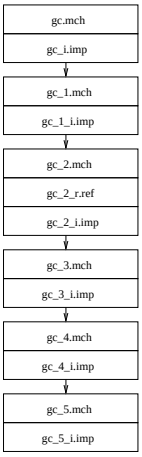

Fig. 3. The B machines dependence graph

assistant, all the proofs have been performed. But the time spent may have been drastically reduced with more experience, mainly in tactics writing. The extensive experience with the Atelier B CASE tool prover has permit to complete quite rapidly the B proof. This study does not permit to obtain a relevant comparison of the cost of the two developments. But taking into account my background, one can consider that the automation of the B prover is not a noticeable advantage.

In terms of readability, the two models should not be seen at the same level. The Coq specification is smaller but it is given at an unique level ; we only present an implementation of the gc and not an abstract model. The B specification presents an abstract and a concrete model, so it is longer. And It contains a lot of repetition which do not help the readability even if the presentation in machine and refinement gives a guide to deal with the model.

Another advantage of the refinement is to structure the lemmas to prove. Properties can be proved at the abstract level and the refinement proof obligations ensure that they are assumed by the implementation. Another benefit of the B method is that the lemmas to prove are automatically produced and the proof is assisted. It seems that, in this study no real problem have been met during the proof in Coq even if the process is very costly since a huge number of proof lines has to be written.

To conclude, the refinement mechanism is helpful to write specification in a hierarchical way, even if the B method force to have a lot of repetition. The Coq representation suffers from a lack of abstraction. But the Coq proof process, even if there is no real automation, is much cleaner that the B one.

References

[Bur00] Lilian Burdy. Traitement des expressions dépourvues de sens de la théorie des ensembles: Application la méthode B. Available from http://cedric.cnam.fr/AfficheArticle.php?id=43, 2000.

[HS97] Klaus Havelund and Natarajan Shankar. A Mechanized Refinement Proof for a Garbage Collector. Available from http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund/, 1997.

[Jac98] Paul B. Jackson. Verifying a Garbage Collection Algorithm. In Jim Grundy and Malcolm Newey, editors, *11th International Conference on Theorem Proving in Higher-Order Logics: TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, September 1998.

[MD99] Luc Moreau and Jean Duprat. A Construction of Distributed Reference Counting. Technical Report RR1999-18, Ecole Normale Supérieure, Lyon, March 1999.

[MT00] Stephanie Motré and Corinne Téri. Using B Method to Formalize the Java Card Runtime Policy for a Common Criteria Evaluation. In *23rd NISSC Proceedings*, October 2000.