A Mechanized Refinement Proof for a Garbage Collector

Klaus Havelund[1]
Dep. of Computer Science
Aalborg University
Frederik Bajersvej 7, 9220 Aalborg Ø, Denmark
havelund@cs.auc.dk
http://www.cs.auc.dk/~havelund
Phone: (+45) 40 28 42 41

Natarajan Shankar[2]
Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
shankar@csl.sri.com
http://www.csl.sri.com/~shankar
Phone: (+1) (415) 859-5272

October 10, 1997

[1]Supported by a European Community HCM grant while working at LITP, Paris 6, France; and by the Danish BRICS project while working at Aalborg University.
[2]Supported by NSF Grant CCR-930044 and by ARPA through NASA Ames Research Center under Contract NASA-NAG-2-891 (ARPA Order A721).

Abstract

We describe how the PVS verification system has been used to verify a safety property of a garbage collection algorithm. The safety property basically says that "nothing but garbage is ever collected". The proof is based on refinement mappings as suggested by Lamport. Although the algorithm is relatively simple, its parallel composition with a "user" program that (nearly) arbitrarily modifies the memory makes the verification quite challenging. The garbage collection algorithm and its composition with the user program is regarded as a concurrent system with two processes working on a shared memory. Such concurrent systems can be encoded in PVS as state transition systems, very similar to the model of, for example, TLA. The safety proof is formulated as a refinement, where the safety specification itself is formulated as state transition system and where the final algorithm is shown to be a refinement thereof. The algorithm is an excellent test-case for formal methods, be they based on theorem proving or model checking. Various hand-written proofs of the algorithm have been developed, some of which are wrong. David Russinoff has verified the algorithm in the Boyer-Moore prover, but his proof is not based on refinement, implying that his safety property cannot be appreciated without a glass box view of the algorithm, considering it's internal structure. Using refinement, however, the algorithm can be regarded as a black box.

Contents

Chapter 1

Introduction

1.1 The Problem

In [12], Russinoff uses the Boyer-Moore theorem prover to verify a safety property of a garbage collection algorithm, originally suggested by Ben-Ari [2]. The safety property is formulated as a predicate P over the state space, and it is verified that this predicate is true in all reachable states. We will describe how the same algorithm can be formulated in the PVS verification system [10], and we demonstrate how the safety property can be verified. However, we shall use a refinement approach, where the safety property itself is formulated as an abstract algorithm, and where the proof is based on refinement mappings as suggested by Lamport. This approach has the advantage that the safety property can be formulated more abstractly without considering the internal structure of the final implementation; the latter being necessary in [12]. Furthermore, as we formalize and mechanize refinement mappings for the general case, this yields a further contribution.

In [7] a proof in PVS is described of the same algorithm and in the invariant-style applied in [12]; the purpose being to directly compare the effort required in PVS compared to the effort required in the Boyer-Moore prover. The conclusion was that the efforts were comparable, which means that PVS can be tuned towards a high degree of automatization due to it's decision procedures.

In [8] we verified a distributed communication protocol using the same techniques for representing state transition systems. Hence, the here presented work shall be seen in the context of a series of PVS verifications of parallel/distributed algorithms, where we try to establish a stabile framework for such verifications, and where we try to identify the main difficulties in carrying out such proofs. The superior goal being to improve the applicability of PVS to this kind of problems. A key conclusion is that techniques for strengthening invariants are of major importance, also in refinement proofs.

The garbage collection algorithm, the *collector*, and its composition with a user program, the *mutator*, is regarded as a concurrent system with (these) two processes working on a shared memory. The memory is basically a structure of nodes, each pointing to other nodes. Some of the nodes are defined as *roots*, which are always accessible to the mutator. Any node that can be reached from a root, chasing pointers, is defined as *accessible* to the mutator. The mutator changes pointers nearly arbitrarily, while the collector continuously collects garbage (not accessible) nodes, and puts them into a *free* list. The collector uses a colouring technique for bookkeeping purposes: each node has a *colour* field associated with it, which is either coloured black if the node is accessible or white if not. In order to make the two processes cooperate correctly, the mutator colours the target node of the redirection black after the redirection. The safety property basically says that *nothing but garbage is ever collected*. Although the collector algorithm is relatively simple, its parallel composition with the mutator makes the verification quite challenging.

In [12], the algorithm is formulated as a state transition system, where the garbage collector at any time is in one of 9 different locations. In one of the locations, say l, an *append* operation is applied to a node n if this node is white, meaning that this node is collected by the collector, assuming it is garbage. The safety property is hence (in principle) formulated as follows:

Whenever in location l, and n is accessible, then n is black

This formulation of the safety property is, however, unfortunate, since it does not really tell us whether the program is correct. We namely in addition have to ensure, that the *append* operation is only called in location l, and only on white nodes. Hence, we need a *glass box* view of the algorithm in order to appreciate the safety property. This observation motivated us to carry out yet an experiment, this time reformulating the safety proof as a refinement, where the safety specification itself is formulated as a state transition system and where the final algorithm is shown to be a refinement thereof. Here a *black box* view of the algorithm is sufficient.

The garbage collector has also been specified and verified in the UNITY framework [4] using a notion of refinement called *superposition*. This refinement notion differs from ours in the sense that the initial algorithm (specification) is not regarded as a specification of lower levels. Rather it is supposed to be an initial algorithm, which is then enriched with more details, until the final enrichment satisfies some (different) property. Each level inherits the properties proved about the previous level, such that the final level inherits all the properties proven for all levels, and this combined collection of properties can the be used to prove, that the final level satisfies some property. In section 8 we shall shortly mention how the UNITY proof differs from ours.

1.2 The History of the Proof

An initial version of the algorithm was first proposed by Dijkstra, Lamport, et al. [6] as an exercise in organizing and verifying the cooperation of concurrent processes. They described their experience as follows:

> Our exercise has not only been very instructive, but at times even humiliating, as we have fallen into nearly every logical trap possible ... It was only too easy to design what looked – sometimes even for weeks and to many people – like a perfectly valid solution, until the effort to prove it correct revealed a (sometimes deep) bug.

Their solution involves three colours. Ben-Ari's later solution is based on the same algorithm, but it only uses two colours, and the proof is therefore simpler. Alternative proofs of Ben-Ari's algorithm were then later published by Van de Snepscheut [5] and Pixley [11]. All of these proofs were informal *pencil and paper* proofs. Ben-Ari defends this as follows:

> So as not to obscure the main ideas, the exposition is limited to the critical facets of the proof. A mechanically verifiable proof would need all sorts of trivial invariants ... and elementary transformations of our invariants (... with appropriate adjustments of the indices).

These four pieces of work, however, indeed show the problem with handwritten proofs, as pointed out by Russinoff [12]; the story goes as follows. Dijkstra, Lamport et al. [6] explained how they (as an example of a "logical trap") originally proposed a modification to the algorithm where the mutator instructions were executed in reverse order (colouring before pointer redirection). This claim was, however, wrong, but was discovered by the authors before the proof reached publication. Ben-Ari then later again proposed this modification and argued for its correctness without discovering its flaw. Counter examples were later given in [11] and [5].

Furthermore, although Ben-Ari's algorithm (which is the one we verify in PVS) is correct, his proof of the safety property was flawed. This flaw was essentially repeated in [11] where it yet again survived the review process, and was only discovered 10 years after when Russinoff detected the flaw during his mechanical proof [12]. As if the story was not illustrative enough, Ben-Ari also gave a proof of a liveness property (*every garbage node will eventually be collected*), and again: this was flawed as later observed in [5]. To put this story of flawed proofs into a context, we shall cite [12]:

> Our summary of the story of this problem is not intended as a negative commentary on the capability of those who have contributed to its solution, all of whom are distinguished scientists. Rather, we present this example as an illustration of the inevitability of human error in the analysis of detailed arguments and as an opportunity to demonstrate the viability of mechanical program verification as an alternative to informal proof.

1.3 Structure of Report

In [13], Shankar demonstrates how concurrent systems can easily be specified in PVS as state transition systems, very similar to the models of, for example, UNITY [4] and TLA [9]. We extend this modeling technique with our own modification of Abadi's and Lamport's refinement mappings [1]; where after we formulate the correctness problem within this refinement framework.

In chapter 2, a formalization of state transition systems and refinement mappings is provided in a loose mathematical style, which is later formalized in PVS. In chapter 3, the garbage collection algorithm is informally described. In chapters 4 and 5, we present the refinements of the algorithm, starting from an initial specification, and ending after three refinement steps with the final algorithm. This presentation is based on an informal notation for writing transition systems. Chapters 6 and 7 formalize in PVS the concepts introduced in earlier chapters 2, 4 and 5. Finally chapter 8 provides some observations on the whole exercise.

Chapter 2

Transition Systems and Refinement Mappings

In this chapter we establish the formal theory for using an abstract non-deterministic program as a safety specification, such that any behaviour is safe if it is generated by the program. An implementation is then defined as a refinement of this program. The basic concepts are those of *transition systems*, *traces*, *invariants*, *observed transition systems*, *refinements*, and *refinement mappings*. The theory presented is a minor modification of the theory developed by Abadi and Lamport, for example as described in [1], and we shall at the end of this chapter outline in which sense we differ. One simplification that we make, and shall mention already here, is to ignore liveness properties and only focus on safety properties. First, we introduce the basic concept of a transition system. Specifications as well as their refinements are written as transition systems.

Definition 2.1 (transition systems) *A transition system is a triple* (Σ, I, N)*, where*

- Σ *is a* state *space*
- $I \subseteq \Sigma$ *is the set of* initial *states.*
- $N \subseteq \Sigma \times \Sigma$ *is the* next-state *relation. Elements of* N *are denoted by pairs of the form* (s, t)*, meaning that there is a transition from the state* s *to the state* t*.*

We shall next define what is an execution trace of a transition system. Informally, an execution trace is an infinite enumeration of states, where the first one satisfies the initiallity predicate and where every two pairs of states are related by the next-state relation. For that purpose we need the notion of a sequence: a sequence σ is an infinite enumeration of states $\langle s_0, s_1, s_2, \ldots \rangle$. We let σ_i denote the i'th element s_i of the sequence. Hence, the traces of a transition system can be defined as follows.

Definition 2.2 (traces) *The traces of a transition system are defined as follows:*

$$\Theta(\Sigma, I, N) = \{\sigma \in \Sigma^* \mid \sigma_0 \in I \land \forall i \geq 0 \cdot N(\sigma_i, \sigma_{i+1})\}$$

We shall need the notion of a transition system invariant, which is a state predicate true in all states reachable from an initial state by following the transition relation.

Definition 2.3 (invariants) *Given a transition system $S = (\Sigma, I, N)$, then a predicate $I : \Sigma \rightarrow \mathcal{B}$ is an S invariant iff.*

$$\forall \sigma \in \Theta(S) \cdot \forall i \geq 0 \cdot I(\sigma_i)$$

Since we want to compare transition systems, and decide whether one transition system refines another, we need a notion of *observability*. For that purpose, we extend transition systems with an *observation function*, which when applied to a state returns an observation in some domain.

Definition 2.4 (observed transition system) *An observed transition system is a five-tuple $(\Sigma, \Sigma_o, I, N, \pi)$ where*

- *(Σ, I, N) is a transition system*
- *Σ_o is a state space, the observed one*
- *$\pi : \Sigma \rightarrow \Sigma_o$ is an* observation function *that extracts the observed part of a state.*

Typically (at least in our case) a state $s \in \Sigma$ consists of an observable part $s_{obs} \in \Sigma_o$ and an internal part s_{int}, hence $s = (s_{obs}, s_{int})$ and π is just the projection function: $\pi(s_{obs}, s_{int}) = s_{obs}$. We adopt the convention that a projection function π applied to a trace $\langle s_1, s_2, \ldots \rangle$ results in the projected trace $\langle \pi_i(s_1), \pi_i(s_2), \ldots \rangle$.

The central concept in all this is the notion of refinement: that one observed transition system S_2 refines another observed transition system S_1. By this we intuitively mean that every observation we can make on S_2, we can also make on S_1. Hence, if S_1 behaves safely so will S_2. This is sufficient when we only want to prove safety properties, as is our case. Put more precisely, it means that every projected trace of S_2 is a projected trace of S_1. This is formulated in the following definition.

Definition 2.5 (refinement) *An observed transition system* $S_2 = (\Sigma_2, \Sigma_o, I_2, N_2, \pi_2)$ refines *an observed transition system* $S_1 = (\Sigma_1, \Sigma_o, I_1, N_1, \pi_1)$ *iff (note that they have the same observed state space* Σ_o*):*

$$\forall t_2 \in \Theta(S_2) \cdot \exists t_1 \in \Theta(S_1) \cdot \pi(t_1) = \pi(t_2)$$

So far, we have established what is an observed transition system, and what it means for one such to refine another such. Hence, the conceptual framework for showing refinement is there. What is missing, is a practical way of showing refinement. Note that refinement is defined in terms of traces. Reasoning about traces is unpractical. What is needed is a way of reasoning about states and pairs of states. *Refinement mappings* is the tool for obtaining this: a refinement mapping from a lower level transition system S_2 to a higher-level one S_1 is a mapping from S_2's state space to S_1's state space, that when applied to (the individual states in) traces, maps traces of S_2 to traces of S_1. This is formally stated as follows.

Definition 2.6 (refinement mapping) *A* refinement mapping *from an observed transition system* $S_2 = (\Sigma_2, \Sigma_o, I_2, N_2, \pi_2)$ *to an observed transition system* $S_1 = (\Sigma_1, \Sigma_o, I_1, N_1, \pi_1)$ *is a mapping* $f : \Sigma_2 \rightarrow \Sigma_1$ *such that there exists an* S_2 *invariant* I*, where:*

1. $\forall s \in \Sigma_2 \cdot \pi_1(f(s)) = \pi_2(s)$

2. $\forall s \in \Sigma_2 \cdot I_2(s) \Rightarrow I_1(f(s))$

3. $\forall s, t \in \Sigma_2 \cdot I(s) \wedge I(t) \wedge N_2(s,t) \Rightarrow N_1(f(s), f(t))$

We can now state the main theorem (which is stated in [1], and which we have proved in PVS for our slightly modified version):

Theorem 2.1 (Existence of Refinement Mappings) *If there exists a refinement mapping from an observed transition system S_2 to an observed transition system S_1, then S_2 refines S_1.*

We shall show how we demonstrate the existence of refinement mappings in PVS, by providing a *witness*, that is: defining a particular one. Defining the refinement mapping turns out typically to be easy, whereas showing that it is indeed a refinement mapping (the properties in definition 2.6) is where the major effort goes. Especially finding the invariant I and prove it, is the bulk of the proof.

We differ from Abadi and Lamport [1] in mainly two ways. First of all, we allow general observation functions, and not just projection functions being the identity on a subsection of the state. In fact, we introduce the notion of observed transition systems, which are five-tuples. This is not explicit in [1]. Second, in definition 2.6 of refinement mappings, we assume that states s and t satisfy an implementation invariant I, which is not the case in [1]. Hence, we have weakened the conditions to prove. Whereas the introduction of observation functions is just a nice (but not strictly necessary) generalization, invariants are of real importance for any practical proofs.

Chapter 3

The Algorithm

In this chapter we informally describe the garbage collection algorithm. As illustrated in figure 3.1, the system consists of two processes, the *mutator* and the *collector*, working on a shared *memory*.

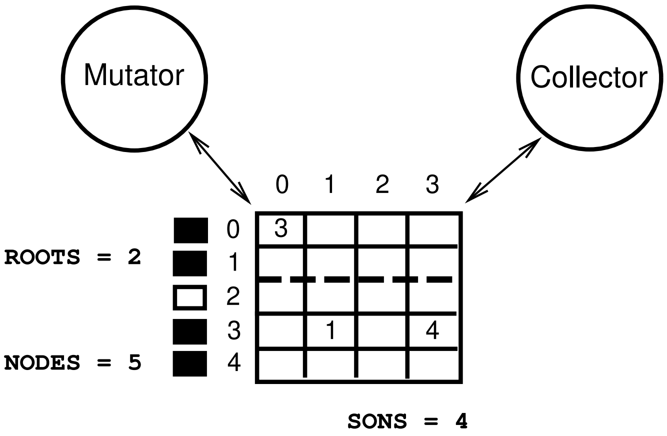

Figure 3.1: The Mutator, Collector and Shared Memory

The Memory

The memory is a fixed size array of *nodes*. In the figure there are 5 nodes (rows) numbered 0 – 4. Associated with each node is an array of uniform length of *cells*. In the figure there are 4 cells per node, numbered 0 – 3. A cell is hence identified by a pair of integers (n,i) where n is a node number and where i is called the *index*. Each cell contains a pointer to a node, called the *son*. In the case of a LISP system, there are for example two cells per node. In the figure we assume that all empty cells contain the *NIL* value 0, hence points to node 0. In addition, node 0 points to node 3 (because cell (0,0) does so), which in turn points to nodes 1 and 4. Hence the memory can be thought of as a two-dimensional array, the size of which is determined by the positive integer constants `NODES` and `SONS`. To each node is associated a *colour*, black or white, which is used by the collector in identifying garbage nodes.

A pre-determined number of nodes, defined by the positive integer constant `ROOTS`, is defined as the *roots*, and these are kept in the initial part of the array (they may be thought of as static program variables). In the figure there are two such roots, separated from the rest with a dotted line. A node is *accessible* if it can be reached from a root by following pointers, and a node is *garbage* if it is not accessible. In the figure nodes 0, 1, 3 and 4 are therefore accessible, and 2 is garbage.

There are only three operations by which the memory structure can be modified:

- Redirect a pointer towards an accessible node.
- Change the colour of a node.
- Append a garbage node to the free list.

In the initial state, all pointers are assumed to be 0, and nothing is assumed about the colours.

The Mutator

The mutator corresponds to the user program and performs the main computation. From an abstract point of view, it continuously changes pointers in the memory; the changes being arbitrary except for the fact that a cell can only be set to point to an already accessible node. In changing a pointer the "previously pointed-to" node may become garbage, if it is not accessible from the roots in some alternative way. In the figure, any cell can hence be modified by the mutator to point to anything else than 2. One should think that only accessible cells could be modified, but the algorithm can in fact be proved safe without that restriction. Hence the less restricted context as possible is chosen. The algorithm is as follows:

1. Select a node n, an index i, and an accessible node k, and assign k to cell (n,i).
2. Colour node k black. Return to step 1.

Each of the two steps are regarded as atomic instructions.

The Collector

The collector's purpose is purely to collect garbage nodes, and put them into a *free list*, from which the mutator may then remove them as they are needed during dynamic storage allocation. Associated with each node is a *colour* field, that is used by the collector during it's identification of garbage nodes. Basically it colours accessible nodes *black*, and at a certain point it collects all *white* nodes, which are then garbage, and puts them into the free list. Figure 3.1 illustrates a situation at such a point: only node 2 is white since only this one is garbage. The collector algorithm is as follows:

1. Colour each root black.
2. Examine each pointer in succession. If the source is black and the target is white, colour the target black.
3. Count the black nodes. If the result exceeds the previous count (or if there was no previous count), return to step 2.
4. Examine each node in succession. If a node is white, append it to the free list; if it is black, colour it white. Then return to step 1.

Steps 1–3 constitute the *marking* phase and their purpose is to blacken all accessible nodes. Each iteration within each step is regarded as an atomic instruction. Hence, for example, step 3 consists of several atomic instructions, each counting (or not) a single node.

The Correctness Criteria

The *safety* property we want to verify is the following: *No accessible node is ever appended to the free list.* In [12], the following *liveness* property is also verified: *Every garbage node is eventually collected.* As in our previous work with a protocol verification in PVS and Murphi [8], we have focused only on safety, since already this requires an effort worth reducing.

Chapter 4

The Specification

In this chapter we give the initial specification of the garbage collector. It's presented as a transition system using an informal notation for describing a such. In chapter 6 it is described how we encode transition systems in PVS.

We shall assume a data structure representing the memory. The number of nodes in the memory is defined by the constant `NODES`. The type `Node` defines the numbers from 0 to `NODES` $-$ 1. The constant `SONS` defines the number of cells per node. The type `Index` defines the numbers from 0 to `SONS` $-$ 1. Hence, the memory can be thought of a two-dimensional array. Figure 4.1 shows the shared state of the specification.

```
var
  M : array[Node,Index] of Node;
```

Figure 4.1: Specification State

The memory will be the observed part of the state (Σ_o – see definition 2.6) throughout all refinements (hence will be the returned value of projection functions). For example, the node colouring structure and other auxiliary variables that we later add will be internal.

Recall, that an initial section of the nodes are roots, the number being defined by the constant `ROOTS`. A number of functions (reading the state) and procedures (modifying the state) are assumed, all of which are mentioned in figure 4.2:

```
function  accessible(n:Node):bool;
function  son(n:Node,i:Index):Node;
procedure set_son(n:Node,i:Index,k:Node);
procedure append_to_free(n:Node);
```

Figure 4.2: Auxiliary Functions used in the Specification

The function `accessible` returns true if it's argument node is accessible from one of the roots by following pointers. The function `son` returns the contents of cell `(n,i)`. The procedure `set_son` assigns `k` to the cell identified by `(n,i)`. Hence after the procedure has been called, this cell now points to `k`. The function `append_to_free` appends it's argument node to the list of free nodes, assuming that it is a garbage node.

The specification consists of the parallel composition of the mutator and the collector. The mutator is described i figure 4.3.

```
MODIFY :
  choose n,k:Node; i:Index where accessible(k) ->
    set_son(n,i,k);
    goto MODIFY
  end
```

Figure 4.3: Specification of Mutator

A program is at any time during its execution considered as being in one of a finite collection of locations, identified by program labels. The above mutator has one such location, named `MODIFY`. Associated with each location is a set of rules, in the basic case each on the form `p -> s` where `p` is a pre-condition on the state and `s` is an effect on the state. When in this location, all rules where the condition `p` is true are enabled, and a non-deterministic choice is made between them, resulting in the next state being obtained by applying the `s` effect of the chosen rule to the current state. The "`choose x:T where p -> s end`" construct represents a set of such rules, one for each choice of `x` within its type `T`. Hence, the mutator repeatedly chooses two arbitrary nodes `n,k:Node` and an arbitrary index `i:Index` such that `k` is accessible. The cell `(n,i)` is then set to point to `k`.

The collector is described in figure 4.4. The collector repeatedly chooses an arbitrary not accessible node. This node is then appended to the free list of nodes. Since the node is not accessible it is a garbage node, hence only garbage nodes are collected (appended), and this is the proper specification of the garbage collector.

```
COLLECT :
  choose n:Node where not accessible(n) ->
    append_to_free(n);
    goto COLLECT
  end
```

Figure 4.4: Specification of Collector

Chapter 5

The Refinement Steps

In this chapter we outline how the refinement is carried out in three steps, resulting in the garbage collection algorithm described informally in chapter 3. Each refinement is given an individual section, which again is divided into a *program* section, presenting the new program, and a *proof* section, outlining the refinement proof performed. According to theorem 2.1 a refinement can be proven by identifying a refinement mapping from the concrete state space to the abstract state space, see definition 2.6. Hence, each *proof* section will consist of a definition of such a mapping together with a proof that it's a refinement mapping, focusing on the simulation relation required in item (3) of definition 2.6.

The PVS encoding of the programs is described in chapter 6, while the PVS encoding of the refinement proofs is described in section 7.

5.1 First Refinement : Introducing Colours

5.1.1 The Program

In the first step, the collector is refined to base it's search for garbage nodes on a colouring technique. The type `Colour` is assumed to contain the two colours *black* and *white*. The global state must be extended with a colouring of each node in the memory (not each cell), and a couple of extra auxiliary variables `Q` and `L` used for other purposes. The extended state is shown in figure 5.1.

```
var
  M : array[Node,Index] of Node;
  C : array[Node] of Colour;
  Q : Node;
  L : nat;
```

Figure 5.1: First Refinement State

```
procedure set_colour(n:Node,c:Colour):void;
function  colour(n:Node):Colour;
function  blackened():bool;
```

Figure 5.2: Additional Auxiliary Functions used in First Refinement

Three extra operations on this new data structure are needed, as indicated in figure 5.2.

The procedure `set_colour` colours a node either white or black. The function `colour` returns the colour of a node. Finally, the function `blackened` returns true if *all accessible nodes are black*.

The mutator is now refined into the program which was informally described in chapter 3, see figure 5.3.

```
MUTATE :
  choose n,k:Nodes; i:Index where accessible(k) ->
    set_son(n,i,k);
    Q := k;
    goto COLOUR;
  end
COLOUR :
  true -> set_colour(Q,black); goto MUTATE;
```

Figure 5.3: Refinement of Mutator

There are two locations, `MUTATE` and `COLOUR`. In the `MUTATE` location, where the mutator starts, in addition to the mutation, the target node `k` is assigned to the global auxiliary variable `Q`. Then in the `COLOUR` location, `Q` is coloured black. *Note that the mutator will not be further refined, it will now stay unchanged during the remaining refinements of the collector.*

The collector is described in figure 5.4. It consists of two phases. While in the `COLOUR` location, nodes are coloured arbitrarily until all accessible nodes are black (`blackened()`). The style in which colouring is expressed may seem surprising, but it is a way of defining a post condition: *colour at least all accessible nodes* [1]. In the second phase, locations `TEST_L` and `APPEND`, all white nodes are regarded as garbage nodes, and are hence collected (appended to the free list). The auxiliary variable `L` is used to control the loop: it runs through all the nodes. After having appended all garbage nodes, the colouring is restarted.

```
COLOUR :
  choose n:Nodes ->
    set_colour(n,black);
    goto COLOUR;
  end
  blackened() -> L := 0; goto TEST_L;
TEST_L :
  L = NODES -> goto COLOUR;
  L < NODES -> goto APPEND;
APPEND :
  not colour(L) -> append_to_free(L); L := L + 1; goto TEST_L;
  colour(L) -> set_colour(L,white); L := L + 1; goto TEST_L;
```

Figure 5.4: First Refinement of Collector

5.1.2 The Refinement Proof

The refinement mapping, call it *abs*, from the concrete state space to the abstract state space maps `M` to `M`. Note that such a mapping only needs to be defined for each component of the abstract state, showing how it is generated from components in the concrete state. Hence, the concrete variables `C`, `Q` and `L` are not used for this purpose. This is generally the case for the refinement mappings to follow: they are the identity on the variables occurring in the abstract state. Also program locations have to be mapped. In

[1] By formulating this colouring as an iteration, we can avoid to introduce a history variable at a lower refinement level.

fact, each program (mutator, collector) can be regarded as having a program counter variable, and we have to show how the abstract program counter is obtained (mapped) from the concrete. Whenever the concrete program is in a particular location l, then the abstract will be in the location $abs(l)$. In the current case, the concrete mutator locations `MUTATE` and `COLOUR` are both mapped to `MODIFY`, while the concrete collector locations `COLOUR`, `TEST_L` and `APPEND` all are mapped to `COLLECT`. This completes the definition of the refinement mapping.

In order to prove property (3) in definition 2.6, we associate each transition in the concrete program with a transition in the abstract program, and prove that: "if the concrete transition brings a state s_1 to a state s_2, then the abstract transition brings the state $abs(s_1)$ to the state $abs(s_2)$". We say that the concrete transition, say t_c, *simulates* the abstract transition, say t_a, and write this as $t_c \ll t_a$. Putting all these sub-proofs together will yield a proof of (3). Some of the concrete transitions just simulate a stuttering step (no state change) in the abstract system. This will typically be some of the new transitions associated with *new location names*, by convention, added to the concrete program. Other concrete transitions have exact counterparts in the abstract program. These are transitions associated with *same location names*, by convention, as in the abstract program. In the following, we will only mention cases that deviate from the above two cases; that is: where we add *new location names*, and where the corresponding transitions do *not* simulate a stuttering step in the abstract program.

Hence in our case, $\texttt{MUTATE} \ll \texttt{MODIFY}$, and $\texttt{APPEND}_1 \ll \texttt{COLLECT}$ ($\texttt{APPEND}_2$ simulates stuttering). In the proof of $\texttt{APPEND}_1 \ll \texttt{COLLECT}$, an invariant is needed about the concrete program:

$$collector@APPEND \wedge accessible(L) \Longrightarrow colour(L) \tag{5.1}$$

It says that whenever the concrete collector is at the `APPEND` location, and node `L` is accessible, then `L` is also black. From this we can conclude that the `append_to_free` operation is only applied to garbage nodes, since it's only applied to white nodes. Hence, we need to prove an invariant about the concrete program in order to prove the refinement. In general, the proof of these invariants is what really makes the refinement proof non-trivial. To prove the above invariant, we do in fact need to prove a stronger invariant, namely that in locations `TEST_L` and `APPEND`, $accessible(n) \Longrightarrow colour(n)$ for all nodes $n \geq L$. This invariant strengthening is typical in our proofs.

5.2 Second Refinement : Colouring by Propagation

5.2.1 The Program

In this step, accessible nodes are coloured through a propagation strategy, where first all roots are coloured, end next all white nodes which have a black father are coloured. The state is extended with an extra auxiliary variable K used for controlling the iteration through the roots. The extended state is shown in figure 5.5.

```
var
  M : array[Node,Index] of Node;
  C : array[Node] of Colour;
  Q : Node;
  K : nat;
  L : nat;
```

Figure 5.5: Second Refinement State

Two additional functions are needed, as indicated in figure 5.6:

```
function bw(n:Node,i:Index):bool;
function exists_bw():bool;
```

Figure 5.6: Additional Auxiliary Functions used in Second Refinement

The function `bw` returns true if `n` is black and `son(n,i)` is white. The function `exists_bw` returns true if there exists a black node, say `n`, that via one of it's cells, say `i`, points to a white node. That is: `bw(n,i)`.

The collector is described in figure 5.7. The `COLOUR` location from the previous level has been replaced by the two locations `COLOUR_ROOTS` and `PROPAGATE` (while the append phase is mostly unchanged). In the `COLOUR_ROOTS` location all roots are coloured black, the loop being controlled by the variable `K`. In the `PROPAGATE` location, either there exists no black node with a white son (i.e. `not exists_bw()`), in which case we start collecting (going to location `TEST_L`), or such a node exists, in which case it's son is coloured black, and we continue colouring.

```
COLOUR_ROOTS :
  K = ROOTS -> goto PROPAGATE;
  K < ROOTS -> set_colour(K,black); K := K+1; goto COLOUR_ROOTS;
PROPAGATE :
  choose n:Node; i:Index where bw(n,i) ->
    set_colour(son(n,i),black);
    goto PROPAGATE;
  end
  not exists_bw() -> L := 0; goto TEST_L;
TEST_L :
  L = NODES -> K := 0; goto COLOUR_ROOTS;
  L < NODES -> goto APPEND;
APPEND :
  not colour(L) -> append_to_free(L); L := L + 1; goto TEST_L;
  colour(L) -> set_colour(L,white); L := L + 1; goto TEST_L;
```

Figure 5.7: Second Refinement of Collector

5.2.2 The Refinement Proof

The refinement mapping, besides being the identity on identically named entities (variables as well as locations), maps the collector locations `COLOUR_ROOTS` and `PROPAGATE` to `COLOUR`. Hence concrete root colouring as well as concrete propagation are just particular kinds of abstract colourings.

Concerning the transitions, $\texttt{COLOUR\_ROOTS}_2 \ll \texttt{COLOUR}_1$, $\texttt{PROPAGATE}_1 \ll \texttt{COLOUR}_1$, and $\texttt{PROPAGATE}_2 \ll \texttt{COLOUR}_2$.

In the proof of $\texttt{PROPAGATE}_2 \ll \texttt{COLOUR}_2$, an invariant is needed about the concrete program:

$$collector@PROPAGATE \Longrightarrow \forall r : Root \cdot colour(r) \tag{5.2}$$

It states that in location `PROPAGATE` all roots must be coloured. This fact combined with the propagation termination condition `not exists_bw()`: "there does not exist a pointer from a black node to a white node", will imply the propagation termination condition in $\texttt{COLOUR}_2$ of the abstract specification: `blackened()`, which says that "all accessible nodes are coloured".

5.3 Third Refinement : Propagation by Scans

5.3.1 The Program

In the last refinement, the propagation, represented by the location PROPAGATE above, is refined into an algorithm, where all nodes are repeatedly scanned in sequential order, and if black, their sons coloured; until a whole scan does not result in a colouring. The collector is described in figure 5.9, where transitions have been divided into 4 steps corresponding to the informal description of the algorithm on page 12. The state is extended with auxiliary variables BC (*black count*) and OBC (*old black count*), used for counting black nodes; and the variables H, I, and J for controlling loops; the state declaration is shown in figure 5.8.

```
var
  M   : array[Node,Index] of Node;
  C   : array[Node] of Colour;
  Q   : Node;
  H   : nat;
  I   : nat;
  J   : nat;
  K   : nat;
  L   : nat;
  BC  : nat;
  OBC : nat;
```

Figure 5.8: Third Refinement State

Two loops interact (steps 2 and 3). In the first loop, TEST_I, TEST_COLOUR and COLOUR_SONS, all nodes are scanned, and every black node has all it's sons coloured. The variables I and J are used to "walk" through the cells. In the second loop, TEST_H, COUNT and COMPARE, it is counted how many nodes are black. This amount is stored in the variable BC, and if this amount exceeds the old black count, stored in the variable OBC, then yet another scan is started, and OBC is updated. The variable H is used to control this loop.

5.3.2 The Refinement Proof

The refinement mapping is the identity, except for six of the locations of the collector. That is, the collector locations `TEST_I`, `TEST_COLOUR`, `COLOUR_SONS`, `TEST_H`, `COUNT`, and `COMPARE` are all mapped to `PROPAGATE`.

Concerning the transitions, $\texttt{COLOUR\_SONS}_2 \ll \texttt{PROPAGATE}_1$ whereas $\texttt{COMPARE}_1 \ll \texttt{PROPAGATE}_2$. In the proof of $\texttt{COLOUR\_SONS}_2 \ll \texttt{PROPAGATE}_1$, the following invariant is needed:

$$collector@COLOUR\_SONS \Longrightarrow colour(I) \tag{5.3}$$

This property implies that the abstract $\texttt{PROPAGATE}_1$ transition's precondition `bw(I,J)` will be true (in case the son is white) or otherwise (if the son is also black), the concrete transition corresponds to a stuttering step (colouring an already black son is the identity function). Correspondingly, in the proof of $\texttt{COMPARE}_1 \ll \texttt{PROPAGATE}_2$, the following invariant is needed:

$$collector@COMPARE \wedge BC = OBC \Longrightarrow \neg exists\_bw() \tag{5.4}$$

It states that when the collector is in location `COMPARE`, after a counting scan where the number of black nodes have been counted and stored in `BC`, if the number counted equals the previous (old) count `OBC` then there does not exist a pointer from a black node to a white node. Note that `BC = OBC` is the propagation termination condition, and this then corresponds to the termination condition `not exists_bw()` of the abstract transition $\texttt{PROPAGATE}_2$.

The proof of these two invariants is quite elaborate, and does in fact compare in size and "look" to the complete proofs in [7] as well as in [12]. Hence, the refinement proof can be said as "containing" these proofs.

```
-- Step 1 : Colour roots

  COLOUR_ROOTS :
    K = ROOTS -> I := 0; goto TEST_I;
    K < ROOTS -> set_colour(K,true); K := K + 1; goto COLOUR_ROOTS;

-- Step 2 : Propagate once

  TEST_I :
    I = NODES -> BC := 0; H := 0; goto TEST_H;
    I < NODES -> goto TEST_COLOUR;
  TEST_COLOUR :
    not colour(I) -> I := I + 1; goto TEST_I;
    colour(I)  -> J := 0; goto COLOUR_SONS;
  COLOUR_SONS :
    J = SONS -> I := I + 1; goto TEST_I;
    J < SONS -> set_colour(son(I,J),black); J := J+1; goto COLOUR_SONS;

-- Step 3 : Count black nodes

  TEST_H :
    H = NODES -> goto COMPARE;
    H < NODES -> goto COUNT;
  COUNT :
    not colour(H) -> H := H +1; goto TEST_H;
    colour(H) -> BC := BC + 1; H := H +1; goto TEST_H;
  COMPARE :
    BC = OBC -> L := 0; goto TEST_L;
    BC /= OBC -> OBC := BD; I := 0; goto TEST_I;

-- Step 4 : Append garbage nodes

  TEST_L :
    L = NODES -> BC := 0; OBC := 0; K := 0; goto TEST_I;
    L < NODES -> goto APPEND;
  APPEND :
    not colour(L) -> append_to_free(L); L := L + 1; goto TEST_L;
    colour(L) -> set_colour(L,white); L := L + 1; goto TEST_L;
```

Figure 5.9: Third Refinement of Collector

Chapter 6

Formalization in PVS

This chapter describes how in general transition systems and refinement mappings are encoded in PVS, and in particular how the garbage collector refinement is encoded. The full set of PVS theories is included in appendix A.

6.1 Transition Systems and their Refinement

Recall from chapter 2 that an observed transition system is a five-tuple of the form: $(\Sigma, \Sigma_o, I, N, \pi)$ (definition 2.4). In PVS we model this as a theory with two type definitions, and three function definitions, see figure 6.1.

The correspondence with the five-tuple is as follows: Σ = `State`, Σ_o = `O_State`, π = `proj`, I = `init` and N = `next`. The `init` function is a predicate on states, while the `next` function is a predicate on pairs of states. We shall formulate the specification of the garbage collector as well as all its refinements in this way. It will become clear below how in particular the function `next` is defined.

Now we can define what is a trace (definition 2.2) and what is an invariant (definition 2.3). This is done in the theory `Traces` in figure 6.2.

The theory is parameterized with the `State` type of the observed transition system. The `VAR` declarations are just associations of types to names, such that in later definitions and axioms, these names are assumed to have the corresponding types. In addition, axioms are assumed to be universally quantified with these names over the types. Note that `pred[T]` in PVS is

```
ots : THEORY

BEGIN

  State   : TYPE = ...
  O_State : TYPE = ...

  proj : [State -> O_State] = ...
  init : [State -> bool] = ...
  next : [State,State -> bool] = ...

END ots
```

Figure 6.1: Observed Transition Systems

```
Traces[State : TYPE] : THEORY

BEGIN

  init : VAR pred[State]
  next : VAR pred[[State,State]]
  sq   : VAR sequence[State]
  n    : VAR nat

  trace(init,next)(sq):bool =
    init(sq(0)) AND
    FORALL n: next(sq(n),sq(n+1))

  p  : VAR pred[State]

  invariant(init,next)(p):bool =
    FORALL (tr:(trace(init,next))): FORALL n: p(tr(n))

END Traces
```

Figure 6.2: Traces and Invariants

short for the function space `[T -> bool]`. The type `sequence[T]` is short for `[nat -> T]`; that is: the set of functions from natural numbers to `T`. A sequence of `State`'s is hence an infinite enumeration of states. Given a transition system with initiality predicate `init` and next-state relation `next`, a sequence `sq` is a trace of this transition system if `trace(init,next)(sq)` holds. A predicate `p` is an invariant if `invariant(init,next)(p)` holds. That is: if for any trace `tr`, `p` holds in all positions `n` of that trace. Note how the predicate `trace(init,next)` (it's a predicate on sequences) is turned into a type in PVS by surrounding it with brackets – the type containing all the elements for which the predicate holds, which are all the traces.

The next notion we introduce in PVS is that of a refinement between two observed transition systems (definition 2.5). Figure 6.3 shows the theory defining the function `refines`, which is a predicate on a pair of observed transition systems: a low level **i**mplementation system as the first parameter, and a high level **s**pecification system as as the second parameter. The theory is parameterized with the state space `S_State` of the high level specification theory, the state space `I_State` of the low level implementation theory, and the observed state space `O_State`, which we remember is common for the two observed transition systems.

Refinement is defined as follows: *for all traces* `i_tr` *of the implementation system, there exists a trace* `s_tr` *of the specification system, such that when mapping the respective projection functions to the traces, they become equal.* The function map has the type `map : [[D->R] -> [sequence[D] -> sequence[R]]]` and simply applies a function to all the elements of a sequence.

Finally, we introduce in the theory `Refinement`, figure 6.4, the notion of a refinement mapping (definition 2.6) and it's use for proving refinement (theorem 2.1). The theory is parameterized with a specification *observed transition system* (prefixes `S`), an implementation *observed transition system* (prefixes `I`), an abstraction function `abs`, and an invariant `I_inv` over the implementation system. The theory contains a number of assumptions on the parameters and a theorem, which has been proven using the assumptions. Hence, the way to use this parameterized theory is to apply it to arguments that satisfy the assumptions, prove these, and then get "as a consequence" the theorem, which states that the implementation refines the specification (corresponding to theorem 2.1). This theorem has been proved once and for all. The assumptions are as stated in definition 2.6.

```
Refine_Predicate[
  O_State : TYPE,
  S_State : TYPE,
  I_State : TYPE] : THEORY

BEGIN

  IMPORTING Traces

  s_init : VAR pred[S_State]
  s_next : VAR pred[[S_State,S_State]]
  s_proj : VAR [S_State -> O_State]

  i_init : VAR pred[I_State]
  i_next : VAR pred[[I_State,I_State]]
  i_proj : VAR [I_State -> O_State]

  refines(i_init,i_next,i_proj)(s_init,s_next,s_proj):bool =
    FORALL (i_tr:(trace(i_init,i_next))):
      EXISTS (s_tr:(trace(s_init,s_next))):
        map(i_proj,i_tr) = map(s_proj,s_tr)

END Refine_Predicate
```

Figure 6.3: Refinement

```
Refinement[
  O_State : TYPE,

  S_State : TYPE,
  S_init  : pred[S_State],
  S_next  : pred[[S_State,S_State]],
  S_proj  : [S_State -> O_State],

  I_State : TYPE,
  I_init  : pred[I_State],
  I_next  : pred[[I_State,I_State]],
  I_proj  : [I_State -> O_State],

  abs     : [I_State -> S_State],
  I_inv   : [I_State -> bool]] : THEORY

BEGIN

  ASSUMING

    IMPORTING Traces

    s     : VAR I_State
    r1,r2 : VAR (I_inv)

    proj_id : ASSUMPTION
      FORALL s: S_proj(abs(s)) = I_proj(s)

    init_h : ASSUMPTION
      FORALL s: I_init(s) IMPLIES S_init(abs(s))

    next_h : ASSUMPTION
      I_next(r1,r2) IMPLIES S_next(abs(r1),abs(r2))

    invar : ASSUMPTION
      invariant(I_init,I_next)(I_inv)

  ENDASSUMING

  IMPORTING Refine_Predicate[O_State,S_State,I_State]

  ref : THEOREM refines(I_init,I_next,I_proj)
                       (S_init,S_next,S_proj)

END Refinement
```

Figure 6.4: Refinement Mappings

```
Refine_Predicate_Transitive[
  O_State : TYPE,
  State1  : TYPE,
  State2  : TYPE,
  State3  : TYPE] : THEORY

BEGIN

  IMPORTING Refine_Predicate

  init1 : VAR pred[State1]
  next1 : VAR pred[[State1,State1]]
  proj1 : VAR [State1 -> O_State]

  init2 : VAR pred[State2]
  next2 : VAR pred[[State2,State2]]
  proj2 : VAR [State2 -> O_State]

  init3 : VAR pred[State3]
  next3 : VAR pred[[State3,State3]]
  proj3 : VAR [State3 -> O_State]

  transitive : LEMMA
    refines[O_State,State2,State3]
      (init3,next3,proj3)(init2,next2,proj2) AND
    refines[O_State,State1,State2]
      (init2,next2,proj2)(init1,next1,proj1)
      IMPLIES
    refines[O_State,State1,State3]
      (init3,next3,proj3)(init1,next1,proj1)

END Refine_Predicate_Transitive
```

Figure 6.5: Refinement is Transitive

We shall further need to assume transitivity of the refinement relation, and this is formulated (and proved) in figure 6.5.

6.2 The Specification

In this chapter we outline how the initial specification from chapter 4 of the garbage collector is modelled in PVS. We start with the specification of the memory structure, and then continue with the two processes that work on this shared structure.

6.2.1 The Memory

The memory type is introduced in a theory, parameterized with the memory boundaries, see figure 6.6. That is, `NODES`, `SONS`, and `ROOTS` define respectively the number of nodes (rows), the number of sons (columns/cells) per node, and the number of nodes that are roots. They must all be positive natural numbers (different from 0). There is also an obvious assumption that `ROOTS` is not bigger than `NODES`. These parameters occur in all our theories.

The `Memory` type is defined as an abstract (non-empty) type upon which a constant and collection of functions are defined[1]. First, however, types of nodes, indexes and roots are defined.

The constant `null_array` represents the initial memory containing `0` in all memory cells (axiom `mem_ax1`). The function `son` returns the pointer contained in a particular cell. That is, the expression `son(n,i)(m)` returns the pointer contained in the cell identified by node `n` and index `i`. Finally, the function `set_son` assigns a pointer to a cell. That is, the expression `set_son(n,i,k)(m)` returns the memory `m` updated in cell `(n,i)` to contain (a pointer to node) `k`.

In order to define what is an accessible node, we introduce the function `points_to`, which defines what it means for one node, `n1`, to point to another, `n2`, in the memory `m`. The function `accessible` is then defined inductively, yielding the least predicate on nodes n (true on the smallest set of nodes) where either n is a root, or n is pointed to from an already reachable node k.

Finally we define the operation for appending a garbage node to the list of free nodes, that can be allocated by the mutator. This operation is defined abstractly, assuming as little as possible about it's behaviour. Note that, since the free list is supposed to be part of the memory, we could easily have defined this operation in terms of the functions `son` and `set_son`, but this would have required that we took some design decisions as to how the

[1]Note that we do not model the memory as a two dimensional array as in chapter 5, since we try to be as abstract as possible during the requirement specification. The reason we used an array earlier was for purely pedagogical reasons.

```
Memory[NODES : posnat, SONS  : posnat, ROOTS : posnat] : THEORY
BEGIN
  ASSUMING roots_within : ASSUMPTION ROOTS <= NODES ENDASSUMING
  IMPORTING List_Functions

  Memory : TYPE+
  Node   : TYPE = {n : nat | n < NODES}
  Index  : TYPE = {i : nat | i < SONS}
  Root   : TYPE = {r : nat | r < ROOTS}

  m         : VAR Memory
  n,n1,n2,k : VAR Node
  i,i1,i2   : VAR Index

  null_array : Memory
  son        : [Node,Index -> [Memory -> Node]]
  set_son    : [Node,Index,Node -> [Memory -> Memory]]

  mem_ax1 : AXIOM son(n,i)(null_array) = 0

  mem_ax2 : AXIOM son(n1,i1)(set_son(n2,i2,k)(m)) =
                  IF n1=n2 AND i1=i2 THEN k ELSE son(n1,i1)(m) ENDIF

  points_to(n1,n2)(m):bool = EXISTS (i:Index): son(n1,i)(m)=n2

  accessible(n)(m): INDUCTIVE bool =
    n < ROOTS OR
    EXISTS k: accessible(k)(m) AND points_to(k,n)(m)

  append_to_free : [Node -> [Memory -> Memory]]
  append_ax: AXIOM
        (NOT accessible(k)(m))
            IMPLIES
          (accessible(n)(append_to_free(k)(m))
                      IFF
          (n = k OR accessible(n)(m)))
END Memory
```

Figure 6.6: The Memory

list was represented (for example where the head of the list should be and whether new elements should be added first or last). The axiom `append_ax` defining the append operation says that *in appending a garbage node, only that node becomes accessible, and the accessibility of all other nodes stay unchanged.*

6.2.2 The Mutator and the Collector

The complete PVS formalization of the garbage collector top level specification presented in chapter 4 is given in figure 6.7.

The state is simply the memory, and so is the observable state. Hence, there are no hidden variables, and the projection function `proj` is the identity. The next-state relation `next` is defined as a disjunction between three disjuncts, each representing a possible single transition of the total system. The first two disjuncts represent a move of the mutator and the collector, respectively, each move defined through a function. The third possibility just represents *stuttering*: the fact that a process does not change the state (needed for technical reasons).

Since each process (mutator, collector) only has one location (see figures 4.3 and 4.4) we don't model these locations explicitly. The function `Rule_mutate` represents a move by the mutator, which is non-deterministic in the choice of the nodes `n,k` and index `i`. The function, when applied to an *old* state, yields a *new* state, where (if `k` is accessible) a pointer has been changed. Non-deterministic choices are modelled via existential quantifications. Each transition function is defined in terms of an `IF-THEN-ELSE` expression, where the condition represents the guard of the transition (the situation where the transition may meaningfully be applied), and where the `ELSE` part returns the unchanged state, in case the guard is false[2]. The function `Rule_append` represents a move by the collector. In each step, either the mutator makes a move, or the collector does. This corresponds to an interleaving semantics of concurrency. Note how the repeated execution is guaranteed by our interpretation of what is a trace in terms of the next-state relation.

[2]This allows for *stuttering* where rules are applied without changing the state. If done infinitely often our system would never progress. One way to avoid such behaviour is to impose certain fairness constraints on execution traces. We shall, however, not do this since we are only interested in verifying safety properties, where such problems play no role. In general, transitions have been modelled as functions, instead of as relations, in order to speed up PVS proofs, using PVS's rewriting engine.

```
Garbage_Collector[NODES : posnat, SONS : posnat, ROOTS : posnat] : THEORY
BEGIN
  ASSUMING roots_within : ASSUMPTION ROOTS <= NODES ENDASSUMING
  IMPORTING Memory[NODES,SONS,ROOTS]

  State   : TYPE = Memory
  O_State : TYPE = Memory

  s,s1,s2 : VAR State
  n,k     : VAR Node
  i       : VAR Index

  proj(s):O_State = s

  init(s):bool = (s = null_array)

  Rule_mutate(n,i,k)(s):State =
    IF accessible(k)(s) THEN
      set_son(n,i,k)(s)
    ELSE s ENDIF

  Rule_append(n)(s):State =
    IF NOT accessible(n)(s) THEN
      append_to_free(n)(s)
    ELSE s ENDIF

  next(s1,s2):bool =
    (EXISTS n,i,k: s2 = Rule_mutate(n,i,k)(s1)) OR
    (EXISTS n: s2 = Rule_append(n)(s1)) OR
    s2 = s1
END Garbage_Collector
```

Figure 6.7: Specification

```
Coloured_Memory[NODES : posnat, SONS : posnat, ROOTS : posnat] : THEORY
BEGIN
  ASSUMING roots_within : ASSUMPTION ROOTS <= NODES ENDASSUMING
  IMPORTING Memory[NODES,SONS,ROOTS]

  Colour  : TYPE = bool
  Colours : TYPE = [Node -> Colour]

  n  : VAR Node
  i  : VAR Index
  c  : VAR Colour
  cs : VAR Colours
  m  : VAR Memory

  colour(n)(cs):Colour = cs(n)

  set_colour(n,c)(cs):Colours = cs WITH [n := c]

  blackened(cs,m):bool = FORALL n: accessible(n)(m) IMPLIES colour(n)(cs)

  bw(n,i)(cs,m):bool = colour(n)(cs) AND NOT colour(son(n,i)(m))(cs)

  exists_bw(cs,m):bool = EXISTS n,i: bw(n,i)(cs,m)
END Coloured_Memory
```

Figure 6.8: Coloured Memory

6.3 The First Refinement

In this section we outline how the first refinement from section 5.1 of the garbage collector is modelled in PVS. In order to keep the presentation reasonably sized, we only illustrate this first refinement. The remaining refinements follow the same pattern and are given in the appendix. First, we describe a collection of colouring functions.

6.3.1 The Coloured Memory

The theory `Coloured_Memory` in figure 6.8 introduces the primitives needed for colouring memory nodes.

The type `Colour` represents the colours *black* (true) and white (false). The type `Colours` contain possible colourings of the memory, each being a mapping from nodes to their colours. The functions `colour`, `set_colour` and `blackened` are formalizations of those presented in figure 5.2. The functions `bw` and `exists_bw` are formalizations of those presented in figure 5.6 and are used in the second refinement.

6.3.2 The Refined Mutator and Collector

We here show how the first refinement is formulated in PVS. The entire theory called `Garbage_Collector1` is presented in figures 6.9 and 6.10. First of all, the state type is a record type with a field for each program variable. In addition to the ordinary program variables, there is a program counter "variable" for each process: `MU` for the mutator, and `CHI` for the collector. Each program counter ranges over a type that contains the possible labels corresponding to those in figures 5.3 and 5.4. The observed state is still just the memory, hence ignoring for example the colouring `C`. We see (figure 6.9) that the mutator next-state relation `MUTATOR` is now defined as a disjunction between a *mutate* transition and a *colour* Q transition. The collector next-state relation `COLLECTOR` (figure 6.10) is defined as the disjunction between six possible transitions.

```
Garbage_Collector1[NODES : posnat, SONS : posnat, ROOTS : posnat] : THEORY
BEGIN
  ASSUMING roots_within : ASSUMPTION ROOTS <= NODES ENDASSUMING
  IMPORTING Coloured_Memory[NODES,SONS,ROOTS]

  MuPC    : TYPE = {MUTATE,COLOUR}
  CoPC    : TYPE = {COLOUR,TEST_L,APPEND}

  State   : TYPE = [# MU : MuPC, CHI : CoPC,
                      Q : nat, L : nat , C : Colours, M : Memory #]

  O_State : TYPE = Memory

  s,s1,s2 : VAR State
  n,k     : VAR Node
  i       : VAR Index

  proj(s):O_State = M(s)

  init(s):bool = MU(s) = MUTATE & CHI(s) = COLOUR & M(s) = null_array

  Rule_mutate(n,i,k)(s):State =
    IF MU(s) = MUTATE AND accessible(k)(M(s)) THEN
      s WITH [M := set_son(n,i,k)(M(s)),
              Q := k,
              MU := COLOUR]
    ELSE s ENDIF

  Rule_colour_target(s):State =
    IF MU(s) = COLOUR AND Q(s) < NODES THEN
      s WITH [C := set_colour(Q(s),TRUE)(C(s)),
              MU := MUTATE]
    ELSE s ENDIF

  MUTATOR(s1,s2):bool =
       (EXISTS n,i,k: s2 = Rule_mutate(n,i,k)(s1))
    OR s2 = Rule_colour_target(s1)
```

Figure 6.9: First Refinement - the Mutator

```
  Rule_stop_colouring(s):State =
    IF CHI(s) = COLOUR AND blackened(C(s),M(s)) THEN
      s WITH [L := 0, CHI := TEST_L]
    ELSE s ENDIF

  Rule_colour(n)(s):State =
    IF CHI(s) = COLOUR THEN
      s WITH [C := set_colour(n,TRUE)(C(s))]
    ELSE s ENDIF

  Rule_stop_appending(s):State =
    IF CHI(s) = TEST_L AND L(s) = NODES THEN
      s WITH [CHI := COLOUR]
    ELSE s ENDIF

  Rule_continue_appending(s):State =
    IF CHI(s) = TEST_L AND L(s) < NODES THEN
      s WITH [CHI := APPEND]
    ELSE s ENDIF

  Rule_black_to_white(s):State =
    IF CHI(s) = APPEND AND L(s) < NODES AND colour(L(s))(C(s)) THEN
      s WITH [C := set_colour(L(s),FALSE)(C(s)),
              L := L(s)+1,
              CHI := TEST_L]
    ELSE s ENDIF

  Rule_append_white(s):State =
    IF CHI(s) = APPEND AND L(s) < NODES AND NOT colour(L(s))(C(s)) THEN
      s WITH [M := append_to_free(L(s))(M(s)),
              L := L(s) + 1,
              CHI := TEST_L]
    ELSE s ENDIF

  COLLECTOR(s1,s2):bool =
       s2 = Rule_stop_colouring(s1)
    OR (EXISTS n: s2 = Rule_colour(n)(s1))
    OR s2 = Rule_stop_appending(s1)
    OR s2 = Rule_continue_appending(s1)
    OR s2 = Rule_black_to_white(s1)
    OR s2 = Rule_append_white(s1)

  next(s1,s2):bool = MUTATOR(s1,s2) OR COLLECTOR(s1,s2) OR s2 = s1
END Garbage_Collector1
```

Figure 6.10: First Refinement - the Collector

Chapter 7

The Proof in PVS

The proof of a single refinement lemma (step) is divided into three activities:

1. Discovery and proof of *function lemmas*
2. Discovery and proof of *invariant lemmas*
3. Proof of the *refinement lemma*

A *function lemma* states a property of one or more auxiliary functions involved, which in our case are for example properties about the functions `accessible` and `blackened`. An invariant is a predicate on states, and an *invariant lemma* states that an invariant holds in every reachable state of the concrete implementation (`Garbage_Collector1` in our case). Recall that we needed such an invariant when applying the `Refinement` theory (figure 6.4). The function lemmas are used in proofs of invariant lemmas, which again are used in proofs of *refinement lemmas*.

We shall show these lemmas for the first refinement, using a bottom-up presentation for pedagogical reasons, starting with function lemmas, and ending with the refinement lemma. In, reality, however, the proof was "discovered" top down: the refinement lemma was stated (by applying the `Refinement` theory to proper arguments), and during the proof of the corresponding `ASSUMPTION`'s, the need for invariant lemmas were discovered, and during their proofs, function lemmas were discovered.

```
Memory_Observers
  [NODES : posnat,
   SONS  : posnat,
   ROOTS : posnat] : THEORY
BEGIN
  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING
  IMPORTING Coloured_Memory[NODES,SONS,ROOTS]

  cs       : VAR Colours
  m        : VAR Memory
  n        : VAR Node
  N,N1,N2 : VAR nat

  blackened(N)(cs,m):bool =
    FORALL (n | N <= n):
      accessible(n)(m) IMPLIES colour(n)(cs)

  ...

END Memory_Observers
```

Figure 7.1: Observer Functions

7.1 Function Lemmas

During the proof, we need a new set of auxiliary functions to "observe" (or calculate) certain values based on the current state of the memory. These *observer functions* occur i invariants. In the first refinement step, we shall need the function `blackened` defined in the theory `Memory_Observers`, see figure 7.1. This function is similar to the function which is part of the first refinement, figure 6.8, except that it has a natural number argument. The function returns true if all nodes above (and including) its argument are black if accessible. The theory contains other functions, but these are first needed in later refinements and will not be discussed here.

The lemmas about auxiliary functions that we need for the first refinement are given in the theory `Memory_Properties` in figure 7.2. The theory in its entirety contains other lemmas, needed for later refinements, which we shall however not present here. The lemma `accessible1` is a key lemma[1], and it says that the `set_son` operator cannot turn garbage nodes into accessible nodes.

7.2 Invariant Lemmas

Invariants are introduced as predicates over the state, and are formulated as being invariants using a collection of special functions, which we have presented in figure 7.3.

The functions `IMPLIES` and `&` are just the corresponding boolean operators lifted to work on state predicates. The function `preserved` allows us to prove a predicate "`p`" as being inductive, assuming the predicate "`inv`" as an induction hypothesis. Using this function, we will be able to split our invariants into manageable sub-lemmas, which in turn can refer to each other in a mutually recursive manner.

We can now state the invariant needed for the first refinement step. This is given in the theory `Garbage_Collector1_Inv` in figure 7.4. The invariant really needed for the refinement proof is `inv1`, corresponding to the invariant (5.1) page 20; but during the proof of that, invariant `inv2` is needed. Invariant `inv1` is in fact the safety property originally formulated for the garbage collector [12]. Its proof requires a generalization, which is `inv2`. This shows an example, where we have to strengthen an invariant (`inv1`) to a stronger invariant (`inv2`), which is then proven instead.

For each new invariant, three declarations are introduced. As an example, the invariant `inv1` gives rise to (1) the definition of the predicate `inv1`, (2) the `i_inv1` lemma, and (3) the `p_inv1` lemma. The latter states the inductive property, while the `i_inv1` lemma makes it possible to refer to `inv1` during the proof of other invariant lemmas.

7.3 The Refinement Lemma

The first refinement step is formulated as an application of the `Refinement` theory which we defined in figure 6.4. This is done in the theory

[1]In appendix A.2 the `accessible1` lemma is in fact introduced in a different special theory `Accessible_Memory_Properties` since it's proof is slightly involved and requires additional definitions and lemmas.

```
Memory_Properties
  [NODES : posnat,
   SONS  : posnat,
   ROOTS : posnat] : THEORY
BEGIN
  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING
  IMPORTING Memory_Observers[NODES,SONS,ROOTS]
  IMPORTING List_Properties

  cs         : VAR Colours
  c          : VAR Colour
  m          : VAR Memory
  n,n1,n2,k  : VAR Node
  i,i1,i2,j  : VAR Index
  N,N1,N2    : VAR nat

  accessible1 : LEMMA
    accessible(k)(m) AND accessible(n1)(set_son(n,i,k)(m))
      IMPLIES accessible(n1)(m)

  blackened1 : LEMMA
    blackened(n)(cs,m) AND accessible(n)(m) IMPLIES colour(n)(cs)

  blackened2 : LEMMA
    accessible(k)(m) AND blackened(N)(cs,m)
      IMPLIES blackened(N)(cs,set_son(n,i,k)(m))

  blackened3 : LEMMA
    blackened(N)(cs,m) IMPLIES blackened(N)(set_colour(n,TRUE)(cs),m)

  blackened4 : LEMMA
    blackened(n)(cs,m) IMPLIES blackened(n+1)(set_colour(n,FALSE)(cs),m)

  blackened5 : LEMMA
    NOT accessible(n)(m) AND blackened(n)(cs,m)
      IMPLIES blackened(n+1)(cs,append_to_free(n)(m))

  blackened6 : LEMMA
    blackened(cs,m) IMPLIES blackened(0)(cs,m)

END Memory_Properties
```

Figure 7.2: Function Lemmas for the First Refinement

```
Invariant_Predicates[State : TYPE] : THEORY

BEGIN

  IMPORTING Traces[State]

  p,p1,p2 : VAR pred[State]
  s,s1,s2 : VAR State
  init    : VAR pred[State]
  next    : VAR pred[[State,State]]
  inv     : VAR pred[State]

  IMPLIES(p1,p2):bool = FORALL s: p1(s) IMPLIES p2(s);

  &(p1,p2):pred[State] = LAMBDA s: p1(s) AND p2(s)

  preserved(init,next)(inv)(p):bool =
    (init IMPLIES p) AND
    FORALL s1,s2:
      inv(s1) AND p(s1) AND next(s1,s2) IMPLIES p(s2)

END Invariant_Predicates
```

Figure 7.3: Invariant Predicates

```
Garbage_Collector1_Inv[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY
BEGIN
  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING
  IMPORTING Memory_Properties[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector1[NODES,SONS,ROOTS]
  IMPORTING Invariant_Predicates[State]

  s : VAR State

  inv1(s):bool =
    CHI(s)=APPEND AND L(s) < NODES AND accessible(L(s))(M(s))
      IMPLIES
    colour(L(s))(C(s))

  inv2(s):bool =
    CHI(s)=TEST_L OR CHI(s)=APPEND IMPLIES blackened(L(s))(C(s),M(s))

  I : pred[State] = inv1 & inv2

  pi : pred[pred[State]] = preserved(init,next)(I)

  i_inv1 : LEMMA I IMPLIES inv1
  i_inv2 : LEMMA I IMPLIES inv2

  p_inv1 : LEMMA pi(inv1)
  p_inv2 : LEMMA pi(inv2)

  p_I : LEMMA pi(I)

  inv : LEMMA invariant(init,next)(I)

END Garbage_Collector1_Inv
```

Figure 7.4: Invariant Lemmas for the First Refinement

```
Refinement1[ NODES : posnat, SONS : posnat, ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  S  : THEORY = Garbage_Collector [NODES,SONS,ROOTS]
  I1 : THEORY = Garbage_Collector1[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector1_Inv[NODES,SONS,ROOTS]

  s     : VAR I1.State
  r1,r2 : VAR (I)
  n,k   : VAR Node
  i     : VAR Index
  cs    : VAR Colours

  abs(s):S.State = M(s)

  ...

  R1 : THEORY =
    Refinement[S.O_State,
      S.State,S.init,S.next,S.proj,
      I1.State,I1.init,I1.next,I1.proj,
      abs,I]

END Refinement1
```

Figure 7.5: First Refinement Lemma

`Refinement1` shown in figure 7.5. The theory imports the specification garbage collector `Garbage_Collector`, giving it the name `S`; the implementation `Garbage_Collector1`, named `I1`; and the implementation invariant `I` defined in the theory `Garbage_Collector1_Inv`. The theory further defines the abstraction function `abs`, and finally applies the `Refinement` theory. This application gives rise to four TCC's (Type Checking Conditions) generated by PVS, which have to be proven in order for the PVS specification to be well formed (type check). Furthermore, the proof of these TCC's yields the correctness of the refinement. The TCC's are shown in figure 7.6.

```
% Assuming TCC generated (line 62) for
    %  Refinement[S.O_State, S.State, S.init, S.next,
    %            S.proj, I1.State, I1.init, I1.next, I1.proj, abs, I]
  % proved - complete
R1_TCC1: OBLIGATION FORALL s: S.proj(abs(s)) = I1.proj(s);

% Assuming TCC generated (line 62) for
    %  Refinement[S.O_State, S.State, S.init, S.next,
    %            S.proj, I1.State, I1.init, I1.next, I1.proj, abs, I]
  % proved - complete
R1_TCC2: OBLIGATION FORALL s: I1.init(s) IMPLIES S.init(abs(s));

% Assuming TCC generated (line 62) for
    %  Refinement[S.O_State, S.State, S.init, S.next,
    %            S.proj, I1.State, I1.init, I1.next, I1.proj, abs, I]
  % proved - complete
R1_TCC3: OBLIGATION
      (FORALL (r1: (I), r2: (I)):
         I1.next(r1, r2) IMPLIES S.next(abs(r1), abs(r2)));

% Assuming TCC generated (line 62) for
    %  Refinement[S.O_State, S.State, S.init, S.next,
    %            S.proj, I1.State, I1.init, I1.next, I1.proj, abs, I]
  % proved - complete
R1_TCC4: OBLIGATION invariant(I1.init, I1.next)(I);
```

Figure 7.6: TCC's Generated by Applying `Refinement`

```
sim_append_white : LEMMA
  r2 = Rule_append_white(r1) IMPLIES
    (EXISTS n: abs(r2) = Rule_append(n)(abs(r1))) OR abs(r2) = abs(r1)
```

Figure 7.7: PVS Version of $\mathtt{APPEND}_1 \ll \mathtt{COLLECT}$

There is a TCC for each ASSUMPTION of the Refinement theory. In particular R1_TCC3 states the simulation property, and R1_TCC4 states the invariant property. As illustrated in section 5.1.2 page 20, we show for each concrete transition which abstract transition it simulates, for example we had that $\mathtt{APPEND}_1 \ll \mathtt{COLLECT}$, which in this PVS setting is formulated as the lemma in figure 7.7.

7.4 Composing the Refinements

The technique illustrated above for the first refinement step is repeated for the next two, yielding two further theories Refinement2 and Refinement3. All 3 refinements can now be composed, and the bottom level implementation can be shown to refine the top level specification using transitivity of the refinement relation. This is expressed in the theory Composed_Refinement in figure 7.8, where the theorem ref is our main correctness criteria.

```
Composed_Refinement[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Refinement1[NODES,SONS,ROOTS]
  IMPORTING Refinement2[NODES,SONS,ROOTS]
  IMPORTING Refinement3[NODES,SONS,ROOTS]

  IMPORTING Refine_Predicate
  IMPORTING Refine_Predicate_Transitive

  ref2 : LEMMA
    refines[S.O_State,S.State,I2.State]
      (I2.init,I2.next,I2.proj)(S.init,S.next,S.proj)

  ref : THEOREM
    refines[S.O_State,S.State,I3.State]
      (I3.init,I3.next,I3.proj)(S.init,S.next,S.proj)

END Composed_Refinement
```

Figure 7.8: Composing the Refinements

Chapter 8

Observations

In this chapter we relate the refinement proof to three other proofs of the same algorithm, two of which are mechanized.

8.1 Comparison with Two Mechanized Proofs

It is possible to compare the present proof with two other mechanized proofs of exactly the same algorithm: a proof in the Boyer-Moore prover [12], from now on referred to as the BM_{inv}-proof; and a PVS proof [7], referred to as the PVS_{inv}-proof. Instead of being based on refinement, these two proofs are based on a statement of the correctness criteria as an invariant to be proven about the implementation (the third refinement step). In addition, a superposition proof has been done in UNITY [4], referred to as the UNITY_{ref}-proof. This proof is manual, hence no theorem prover has been used.

The PVS_{inv}-proof follows the BM_{inv}-proof closely. Basically the same invariants were used, but the PVS_{inv}-proof used 65 function lemmas, whereas the BM_{inv}-proof used over 100. The BM-lemmas were not published, so we have not been able to examine the reason for this difference. One reason could be that our lemmas are more general.

The PVS_{ref}-proof has the advantage over the two other proofs, that the correctness criteria can be appreciated without knowing the internal structure of the implementation. That is, we do not need to know for example that the append operation is only applied in location `APPEND` to node `L`, and only if `L` is white. Hence, from this perspective, the refinement proof represents an improvement.

The PVS_{ref}-proof has approximately the same size as the PVS_{inv}-proof, in that basically the same invariants and lemmas about auxiliary functions need to be proven. Hence, one cannot argue that the proof has become any simpler. On the contrary in fact: since we have many levels, we have to prove some invariants several times.

One can perhaps say, that some invariants were easier to discover when using refinement, especially at the top levels. In particular nested loops may be treated nicely with refinement, only introducing one loop at a time. In general, loops in the algorithm to be verified are the reason why invariant discovery is hard, and of course nested loops are no better.

The main lesson obtained from the PVS_{inv}-proof was that invariant discovery was the key element to focus attention on. The experience with the PVS_{ref}-proof is in fact the same: refinement does not relieve us from searching invariants. We had to come up with exactly the same invariants, but the discovery process was different, perhaps more structured.

Our lesson can be summarized in the following formula, where S is a transition system, and where P and Q are state predicates, and $\Box X$ (X being P or Q) means *always* X, and where $S \vdash \Box X$ means that X is true in all reachable states of S, hence: X is an S invariant.

$$\frac{S \vdash \Box P, P \Rightarrow Q}{S \vdash \Box Q}$$

The formula says that if we want to prove $S \vdash \Box Q$, we have to find a stronger predicate P (which implies Q), and prove $S \vdash \Box P$. Now proving $S \vdash \Box P$, once we have P is normally not the problem, assuming a good theorem prover like PVS. Here we believe that the decision procedures in PVS helped us greatly.

The problem is to find P. This is no new knowledge, since it corresponds to finding the loop-invariant in while programs, and this is known to be unsolvable in general. However, heuristics for finding P in many practical situations may be feasible. A second lesson is that even using refinement does not relieve us from the invariant discovery process. Hence, it is central. The problem has also been investigated in [3] by Saddek Bensalem, amongst others.

8.2 Comparison with the Manual UNITY Proof

In [4] a proof has been carried out of basically the same algorithm using the technique of superposition. We describe that effort in the following, comparing it to our proof.

8.2.1 The Superposition Technique

The UNITY$_{ref}$-proof is based on superposition. As mentioned earlier, this notion of refinement differs from ours in the sense that the initial program is not regarded as a specification of lower levels. Rather it is supposed to be just a starting point, called the *underlying program* which is then enriched with more details, in several steps, until the final enrichment, call it the *goal program* satisfies some (different) property, which we shall call the *goal property*. Hence, this kind of proof is in fact closer to the other invariant proofs PVS$_{inv}$ and PVS$_{ref}$ mentioned above, in that what we really end up with is a goal program, which then must satisfy some goal property, which could for example be an invariant. The stepwise manner in which this goal program is developed does of course resemble refinement, but it's rather a horizontal refinement than a vertical one. Each level in the enrichment chain inherits the properties proven about the previous level, and all the collected properties can then be used to prove that the goal program satisfies the goal property.

8.2.2 Overview of the Proof

The UNITY program to be developed is planned to consist in principle of a mutator in parallel with a garbage collector. They both work on a graph with a fixed set of vertices where edges are changed by the mutator. One of the vertices is the root, and any vertice reachable from the root is accessible. The mutator works very much the same way as ours. The garbage collector moves garbage vertices into the free list. Each execution of the garbage collector consists of an execution of a program called *marker* followed by an execution of a program called *collector*. Program *marker* marks all accessible vertices; where after the *collector* places unmarked vertices on the free list. The algorithm verified is in fact only the *marker* in parallel with the mutator, leaving out the *collector*. Hence, a simplification compared to our proof. Another simplification is, that the *marker* is only called once, and not repeatedly as it is in our model.

8.2.3 The Initial Program

The initial underlying program (to be enriched) in the UNITY$_{ref}$-proof just consists of the mutator, adding and deleting edges between vertices. Hence, already here the approach differs, since our initial program (section 4) is in fact a complete specification of the final program, which includes this mutator (figure 4.3), but also an abstract version of the garbage collector (figure 4.4), that appends garbage vertices to the free list, without specifying how garbage vertices are identified.

8.2.3.1 The First Superposition

This program is then enriched in two steps. In the first step, a program *propagator* is obtained, which (in addition to the mutator's activity) contains a first development of the garbage collector, which repeatedly marks some arbitrary vertice that is pointed to from an already marked vertice (propagation of marked vertices). Prior to the execution, the root is marked. Hence, this should sooner or later result in the marking of all accessible vertices. In addition, when the mutator adds an arc, the target is marked, just as in our own case.

The *propagator* should be compared to our second refinement (section 5.2), the mutator of which is presented in figure 5.3, and the collector presented in figure 5.7. There are three differences. First, the UNITY$_{ref}$ mutator marks the target simultaneously with the addition of the edge, while in our case, the target is assigned to the variable Q, and then Q is marked in the mutator's next transition. Second, the UNITY$_{ref}$ program operates with one root while we operate with several. Third, our second refinement contains a condition for terminating the propagation, namely `not exists_bw()`: *there does not exist a pointer from a black node to a white node.* This condition is not part of the second UNITY$_{ref}$ refinement, which just continues marking. In fact, the third UNITY$_{ref}$ superposition, to be described below, has as purpose to add this termination condition. As we, see, this again illustrates the difference in approach: our refinement contains a complete view of the system, while the UNITY$_{ref}$ refinement only provides a partial view (without a termination condition).

8.2.3.2 The Second Superposition

The final program, the *marker* is obtained by adding a termination condition. This is done by adding four variables: two boolean flags, *mu.flag* and *ma.flag*, one for the mutator and one for the garbage collector; a variable b, containing a set of edges, and a boolean variable *over*, which will become *true* eventually when all accessible vertices have been marked. The variable b is initially the empty set, and *over* is *false* (the two flags have arbitrary values). The final superposition is now done by modifying the mutator and the garbage collector, and by adding three new, always enabled, transitions which manipulate the four new variables.

The mutator and garbage collector are modified such that they set their respective flags to *false* in case they mark a non-marked vertice. The three new transitions are as follows. First, a transition simply adds an edge (u,v) to b in case it holds that: "*marked(u)* implies *marked(v)*". Second, a transition empties b and assigns *true* to the two flags in case any of them are *false*. Finally, a third transition assigns *true* to the variable *over* in case both flags are *true* and also the variable b contains all possible edges.

This program differs from our third and final refinement (section 5.3) presented in figure 5.9 in basically two ways. First, we have refined the propagation further such that nodes are scanned in a pre-defined lexicographic order, and not arbitrarily as in the UNITY_{ref} program. Second, we model the termination condition by another scan of all nodes, counting the number of black nodes, and comparing this number to the previous count, terminating the propagation only if these numbers are equal (no new nodes have been coloured). If they are not equal, a lexicographically ordered propagation is repeated.

8.2.4 The Proof

The property proven about the final *marker* is:

- The variable *over* eventually becomes *true*
- When *over* is *true*, all accessible nodes are marked.

This is a liveness property in contrast to ours, which is a safety property. Our safety property is basically stated in figure 4.4, and states that only garbage nodes are appended to the free list. Since the UNITY$_{ref}$ program does not include this *collecter* phase, the correctness property must be stated differently, hence as a liveness property.

Acknowledgments Saddek Bensalem (VERIMAG) has been involved in quite detailed discussions about invariant discovery. Sam Owre (SRI) has assisted with the use of PVS. John Rushby (SRI) has been influential in his encouragement and his role in setting up the collaboration. Finally, Therese Hardin (LITP) provided a comfortable environment during Klaus Havelund's stay at LITP, Paris 6, France.

Appendix A

PVS Theories

A.1 Transition Systems and their Refinement

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Traces :                                                     %
%   Assuming a transition system (State,init,next) this theory %
%   defines what a trace is and what an invariant is.          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Traces[State : TYPE] : THEORY

BEGIN

  init : VAR pred[State]
  next : VAR pred[[State,State]]
  sq   : VAR sequence[State]
  n    : VAR nat

  trace(init,next)(sq):bool =
    init(sq(0)) AND
    FORALL n: next(sq(n),sq(n+1))

  p  : VAR pred[State]

  invariant(init,next)(p):bool =
    FORALL (tr:(trace(init,next))): FORALL n: p(tr(n))

END Traces
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Refine_Predicate :                                      %
%   Defines the predicate ``refines'' expressing when one %
%   transition system refines another.                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Refine_Predicate[
  O_State : TYPE,
  S_State : TYPE,
  I_State : TYPE] : THEORY

BEGIN

  IMPORTING Traces

  s_init : VAR pred[S_State]
  s_next : VAR pred[[S_State,S_State]]
  s_proj : VAR [S_State -> O_State]

  i_init : VAR pred[I_State]
  i_next : VAR pred[[I_State,I_State]]
  i_proj : VAR [I_State -> O_State]

  refines(i_init,i_next,i_proj)(s_init,s_next,s_proj):bool =
    FORALL (i_tr:(trace(i_init,i_next))):
      EXISTS (s_tr:(trace(s_init,s_next))):
        map(i_proj,i_tr) = map(s_proj,s_tr)

END Refine_Predicate
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Refine_Predicate_Transitive :                      %
%   States as a lemma that refinement is transitive. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Refine_Predicate_Transitive[
  O_State : TYPE,
  State1  : TYPE,
  State2  : TYPE,
  State3  : TYPE] : THEORY

BEGIN

  IMPORTING Refine_Predicate

  init1 : VAR pred[State1]
  next1 : VAR pred[[State1,State1]]
  proj1 : VAR [State1 -> O_State]

  init2 : VAR pred[State2]
  next2 : VAR pred[[State2,State2]]
  proj2 : VAR [State2 -> O_State]

  init3 : VAR pred[State3]
  next3 : VAR pred[[State3,State3]]
  proj3 : VAR [State3 -> O_State]

  transitive : LEMMA
    refines[O_State,State2,State3]
      (init3,next3,proj3)(init2,next2,proj2) AND
    refines[O_State,State1,State2]
      (init2,next2,proj2)(init1,next1,proj1)
      IMPLIES
    refines[O_State,State1,State3]
      (init3,next3,proj3)(init1,next1,proj1)

END Refine_Predicate_Transitive
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Refinement :                                              %
%   The lemma ``ref'' states that the implementation refines %
%   the specification conditioned the assumptions.          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Refinement[
  O_State : TYPE,

  S_State : TYPE,
  S_init  : pred[S_State],
  S_next  : pred[[S_State,S_State]],
  S_proj  : [S_State -> O_State],

  I_State : TYPE,
  I_init  : pred[I_State],
  I_next  : pred[[I_State,I_State]],
  I_proj  : [I_State -> O_State],

  abs     : [I_State -> S_State],
  I_inv   : [I_State -> bool]] : THEORY

BEGIN

  ASSUMING

    IMPORTING Traces

    s     : VAR I_State
    r1,r2 : VAR (I_inv)

    proj_id : ASSUMPTION
      FORALL s: S_proj(abs(s)) = I_proj(s)

    init_h : ASSUMPTION
      FORALL s: I_init(s) IMPLIES S_init(abs(s))

    next_h : ASSUMPTION
      I_next(r1,r2) IMPLIES S_next(abs(r1),abs(r2))

    invar : ASSUMPTION
      invariant(I_init,I_next)(I_inv)

  ENDASSUMING

  IMPORTING Refine_Predicate[O_State,S_State,I_State]
```

```
  ref : THEOREM refines(I_init,I_next,I_proj)
                       (S_init,S_next,S_proj)

END Refinement
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Invariant_Predicates :                   %
%   Functions used for proving invariants. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Invariant_Predicates[State : TYPE] : THEORY

BEGIN

  IMPORTING Traces[State]

  p,p1,p2 : VAR pred[State]
  s,s1,s2 : VAR State
  init    : VAR pred[State]
  next    : VAR pred[[State,State]]
  inv     : VAR pred[State]

  IMPLIES(p1,p2):bool = FORALL s: p1(s) IMPLIES p2(s);

  &(p1,p2):pred[State] = LAMBDA s: p1(s) AND p2(s)

  preserved(init,next)(inv)(p):bool =
    (init IMPLIES p) AND
    FORALL s1,s2:
      inv(s1) AND p(s1) AND next(s1,s2) IMPLIES p(s2)

  preserved_and : LEMMA
    preserved(init,next)(inv)(p1) AND
    preserved(init,next)(inv)(p2)
      IMPLIES
    preserved(init,next)(inv)(p1 & p2)

  preserved_inv : LEMMA
    preserved(init,next)(inv)(inv) IMPLIES invariant(init,next)(inv)

END Invariant_Predicates
```

A.2 The Memory

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Memory :                                                     %
%   Defines the memory type and the basic operations upon it. %
%   Defines what it means for a node to be accessible in a    %
%   memory.                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Memory[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  Memory : TYPE+

  Node    : TYPE = {n : nat | n < NODES}
  Index   : TYPE = {i : nat | i < SONS}
  Root    : TYPE = {r : nat | r < ROOTS}

  m : VAR Memory
  n,n1,n2,k : VAR Node
  i,i1,i2 : VAR Index

  null_array : Memory
  son        : [Node,Index -> [Memory -> Node]]
  set_son    : [Node,Index,Node -> [Memory -> Memory]]

  mem_ax1 : AXIOM son(n,i)(null_array) = 0

  mem_ax2 : AXIOM son(n1,i1)(set_son(n2,i2,k)(m))
                    =
                  IF n1=n2 AND i1=i2 THEN k ELSE son(n1,i1)(m) ENDIF

  points_to(n1,n2)(m):bool =
    EXISTS (i:Index): son(n1,i)(m)=n2

  accessible(n)(m): INDUCTIVE bool =
    n < ROOTS OR
    EXISTS k: accessible(k)(m) AND points_to(k,n)(m)

  append_to_free : [Node -> [Memory -> Memory]]
```

```
  append_ax: AXIOM
        (NOT accessible(k)(m))
            IMPLIES
          (accessible(n)(append_to_free(k)(m)) IFF (n = k OR accessible(n)(m)))

END Memory
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Accessible_Memory_Properties :                     %
%   States lemmas about the ``accessible'' function. %
%   These lemmas are needed during invariant proofs. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Accessible_Memory_Properties[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Memory[NODES,SONS,ROOTS]

  m : VAR Memory
  n,n1,n2,k,k1 : VAR Node
  i : VAR Index
  l,l1,l2 : VAR list[Node]

  path(m)(n, n1)(l): RECURSIVE bool =
    (CASES l OF
       null: n = n1,
       cons(k,l): (points_to(k, n1)(m) AND
                   path(m)(n, k)(l))
     ENDCASES)
     MEASURE length(l)

  member_cdr(n, l): RECURSIVE list[Node] =
     (CASES l OF
        null: null,
        cons(k, l1): (IF k = n THEN l1
                      ELSE member_cdr(n, l1) ENDIF)
      ENDCASES)
      MEASURE length(l)

  set_son_points_to_1: LEMMA
      points_to(n, k)(set_son(n, i, k)(m))

  set_son_points_to_2: LEMMA
      n /= n1 IMPLIES
        points_to(n1, k1)(set_son(n, i, k)(m)) =
        points_to(n1, k1)(m)
```

```
set_son_points_to_3: LEMMA
    k/=k1 AND
    points_to(n1, k1)(set_son(n, i, k)(m))
    IMPLIES  points_to(n1, k1)(m)

path_append: LEMMA
   path(m)(n, n1)(l1) AND path(m)(k, k1)(l2) AND points_to(k1, n)(m)
     IMPLIES path(m)(k, n1)(append(l1, cons(k1, l2)))

accessible_path: LEMMA
  accessible(n)(m)
    = (EXISTS n1, l: path(m)(n1, n)(l) AND n1 < ROOTS)

path_member_cdr: LEMMA
  path(m)(n, n1)(l) AND member(k, l) IMPLIES
    path(m)(n, k)(member_cdr(k, l))

length_member_cdr: LEMMA
  cons?(l) IMPLIES
    length(member_cdr(n, l)) < length(l)

path_without_duplicates: LEMMA
  path(m)(n, n1)(l) IMPLIES
    (EXISTS l1: path(m)(n, n1)(l1) AND
                 NOT member(n1, l1))

path_set_son: LEMMA
  path(m)(n, n1)(l) AND NOT member(k, l) IMPLIES
    path(set_son(k, i, k1)(m))(n, n1)(l)

path_set_son_2: LEMMA
  path(set_son(k, i, k1)(m))(n, n1)(l) AND
  NOT member(k, l)
    IMPLIES
  path(m)(n, n1)(l)

path_set_son_3: LEMMA
  path(set_son(k, i, k1)(m))(n, n1)(l) AND
  path(m)(n2, k1)(l2)
    IMPLIES
  (EXISTS l1: path(m)(n, n1)(l1) OR
              path(m)(n2, n1)(l1))

accessible1 : LEMMA
  accessible(k)(m) AND accessible(n1)(set_son(n,i,k)(m))
    IMPLIES
```

```
    accessible(n1)(m)

  accessible2 : LEMMA
    accessible(k)(m) IMPLIES accessible(k)(set_son(n,i,k)(m))

END Accessible_Memory_Properties
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Coloured_Memory :                                         %
%   Defines functions to colour (and examine the colours of) %
%   memory nodes.                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Coloured_Memory[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Memory[NODES,SONS,ROOTS]

  Colour  : TYPE = bool
  Colours : TYPE = [Node -> Colour]

  n  : VAR Node
  i  : VAR Index
  c  : VAR Colour
  cs : VAR Colours
  m  : VAR Memory

  colour(n)(cs):Colour =
    cs(n)

  set_colour(n,c)(cs):Colours =
    cs WITH [n := c]

  blackened(cs,m):bool =
    FORALL n: accessible(n)(m) IMPLIES colour(n)(cs)

  bw(n,i)(cs,m):bool =
    colour(n)(cs) AND NOT colour(son(n,i)(m))(cs)

  exists_bw(cs,m):bool =
    EXISTS n,i: bw(n,i)(cs,m)

END Coloured_Memory
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Memory_Observers :                                       %
%   Defines memory observers, being functions that extract %
%   information from the memory. These functions are used  %
%   when stating the invariants.                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Memory_Observers
  [NODES : posnat,
   SONS  : posnat,
   ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Coloured_Memory[NODES,SONS,ROOTS]

  cs       : VAR Colours
  m        : VAR Memory
  n        : VAR Node
  i        : VAR Index
  r        : VAR Root
  N,N1,N2  : VAR nat
  I1,I2    : VAR nat
  cl1,cl2  : VAR [nat,nat]

  blackened(N)(cs,m):bool =
    FORALL (n | N <= n):
      accessible(n)(m) IMPLIES colour(n)(cs)

  black_roots(N)(cs):bool =
    FORALL (r | r < N): colour(r)(cs);

  <(cl1,cl2):bool =
    LET
      n1 = PROJ_1(cl1), i1 = PROJ_2(cl1),
      n2 = PROJ_1(cl2), i2 = PROJ_2(cl2)
    IN
      n1 < n2 OR (n1 = n2 AND i1 < i2);

  exists_bw(N1,I1,N2,I2)(cs,m):bool =
    EXISTS n,i:
      bw(n,i)(cs,m) AND
      NOT (n,i) < (N1,I1) AND
```

```
      (n,i) < (N2,I2)

  blacks(N1,N2)(cs) : RECURSIVE nat =
    IF N1 < N2 AND N1 < NODES THEN
      IF colour(N1)(cs) THEN 1 ELSE 0 ENDIF + blacks(N1+1,N2)(cs)
    ELSE
      0
    ENDIF
    MEASURE abs(N2-N1)

END Memory_Observers
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Memory_Properties :                               %
%   States lemmas about memory observers.           %
%   These lemmas are needed during invariant proofs. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Memory_Properties[NODES: posnat, SONS: posnat, ROOTS: posnat]: THEORY
BEGIN

  ASSUMING
    roots_within: ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Memory_Observers[NODES, SONS, ROOTS]

  IMPORTING Accessible_Memory_Properties[NODES, SONS, ROOTS]

  abs((i: int)): nat = IF i < 0 THEN -i ELSE i ENDIF

  cs : VAR Colours
  c : VAR Colour
  m : VAR Memory
  n, n1, n2, k : VAR Node
  i, i1, i2, j : VAR Index
  N, N1, N2 : VAR nat
  I, I1, I2 : VAR nat
  x : VAR nat
  l, l1, l2 : VAR list[Node]

  smaller1 : LEMMA
    NOT (n, i) < (0, 0)

  smaller2 : LEMMA
    (NOT (n, i) < (k, 0) AND (n, i) < (k + 1, 0)) IMPLIES n = k

  smaller3 : LEMMA
    (n, i) < (k, SONS) IFF (n, i) < (k + 1, 0)

  smaller4 : LEMMA
    (NOT (n, i) < (k, j) AND (n, i) < (k, j + 1)) IMPLIES (n, i) = (k, j)

  colour1 : LEMMA
    colour(n)(set_colour(n1, c)(cs)) = IF n = n1 THEN c ELSE colour(n)(cs) ENDIF

  colour2 : LEMMA
    colour(n)(cs) IMPLIES set_colour(n, TRUE)(cs) = cs
```

```
blackened1 : LEMMA
  blackened(n)(cs, m) AND accessible(n)(m) IMPLIES colour(n)(cs)

blackened2 : LEMMA
  accessible(k)(m) AND blackened(N)(cs, m) IMPLIES blackened(N)(cs, set_son(n, i, k)(m))

blackened3 : LEMMA
  blackened(N)(cs, m) IMPLIES blackened(N)(set_colour(n, TRUE)(cs), m)

blackened4 : LEMMA
  blackened(n)(cs, m) IMPLIES blackened(n + 1)(set_colour(n, FALSE)(cs), m)

blackened5: LEMMA
  NOT accessible(n)(m) AND blackened(n)(cs, m) IMPLIES
    blackened(n + 1)(cs, append_to_free(n)(m))

blackened6 : LEMMA
  blackened(cs, m) IMPLIES blackened(0)(cs, m)

blackened7 : LEMMA
  black_roots(ROOTS)(cs) AND NOT exists_bw(cs, m) IMPLIES blackened(cs, m)

blackened8 : LEMMA
  blackened(cs, m) AND accessible(n)(m) IMPLIES colour(n)(cs)

black_roots1 : LEMMA
  black_roots(N)(cs) IMPLIES black_roots(N)(set_colour(n, TRUE)(cs))

black_roots2 : LEMMA
  black_roots(0)(cs)

black_roots3 : LEMMA
  black_roots(n)(cs) IMPLIES black_roots(n + 1)(set_colour(n, TRUE)(cs))

bw1 : LEMMA
  bw(n, i)(cs, m) = (colour(n)(cs) AND NOT colour(son(n, i)(m))(cs))

bw2 : LEMMA
  (NOT bw(n1, i1)(cs, m) AND bw(n1, i1)(cs, set_son(n2, i2, k)(m)))
    IMPLIES
  (n1, i1) = (n2, i2)

bw3: LEMMA
  (NOT bw(n, i)(cs, m) AND bw(n, i)(set_colour(k, TRUE)(cs), m))
    IMPLIES
  (n = k AND NOT colour(n)(cs))
```

```
bw4 : LEMMA
  bw(n, i)(cs, m) IMPLIES colour(n)(cs) AND NOT colour(son(n, i)(m))(cs)

exists_bw1 : LEMMA
  colour(k)(cs) AND NOT exists_bw(cs, m) IMPLIES NOT exists_bw(cs, set_son(n, i, k)(m))

exists_bw2 : LEMMA
  exists_bw(cs, m) = exists_bw(0, 0, NODES, 0)(cs, m)

exists_bw3 : LEMMA
  NOT exists_bw(N, I, N, I)(cs, m)

exists_bw4 : LEMMA
  NOT exists_bw(0, 0, N, I)(cs, m) AND exists_bw(0, 0, N, I)(cs, set_son(n, i, k)(m))
    IMPLIES
  NOT colour(k)(cs) AND (n, i) < (N, I)

exists_bw5 : LEMMA
  accessible(n)(m) AND NOT colour(n)(cs) AND black_roots(ROOTS)(cs)
    IMPLIES
  exists_bw(0, 0, NODES, 0)(cs, m)

exists_bw6 : LEMMA
  exists_bw(0, 0, NODES, 0)(cs, m)
    IMPLIES
  exists_bw(0, 0, N, I)(cs, m) OR exists_bw(N, I, NODES, 0)(cs, m)

exists_bw7 : LEMMA
  exists_bw(N, I, NODES, 0)(cs, m) AND (n, i) < (N, I)
    IMPLIES
  exists_bw(N, I, NODES, 0)(cs, set_son(n, i, k)(m))

exists_bw8 : LEMMA
  NOT colour(n)(cs) AND exists_bw(0, 0, n + 1, 0)(cs, m)
    IMPLIES
  exists_bw(0, 0, n, 0)(cs, m)

exists_bw9 : LEMMA
  NOT colour(n)(cs) AND exists_bw(n, 0, NODES, 0)(cs, m)
    IMPLIES
  exists_bw(n + 1, 0, NODES, 0)(cs, m)

exists_bw10 : LEMMA
  exists_bw(0, 0, N + 1, 0)(cs, m) IMPLIES exists_bw(0, 0, N, SONS)(cs, m)

exists_bw11 : LEMMA
  exists_bw(N, SONS, NODES, 0)(cs, m) IMPLIES exists_bw(N + 1, 0, NODES, 0)(cs, m)
```

```
  exists_bw12 : LEMMA
    colour(son(n, i)(m))(cs) AND exists_bw(0, 0, n, i + 1)(cs, m)
      IMPLIES
    exists_bw(0, 0, n, i)(cs, m)

  exists_bw13 : LEMMA
    colour(son(n, i)(m))(cs) AND exists_bw(n, i, NODES, 0)(cs, m)
      IMPLIES
    exists_bw(n, i + 1, NODES, 0)(cs, m)

  exists_bw14 : LEMMA
    exists_bw(N1, I1, N2, I2)(cs, m)
      IMPLIES
    EXISTS (n: Node, i: Index):
      bw(n, i)(cs, m) AND NOT (n, i) < (N1, I1) AND (n, i) < (N2, I2)

  blacks1 : LEMMA
    blacks(N, N)(cs) = 0

  blacks2 : LEMMA
    blacks(N1, N2)(cs) <= blacks(N1, N2)(set_colour(n, TRUE)(cs))

  blacks3 : LEMMA
    NOT colour(n)(cs) IMPLIES blacks(n, N)(cs) = blacks(n + 1, N)(cs)

  blacks4 : LEMMA
    (n < N AND colour(n)(cs)) IMPLIES blacks(n, N)(cs) = blacks(n + 1, N)(cs) + 1

  blacks5_1 : LEMMA
    (n < N1 OR n >= N2) IMPLIES blacks(N1, N2)(set_colour(n, c)(cs)) = blacks(N1, N2)(cs)

  blacks5_2 : LEMMA
    (n >= N1 AND n < N2 AND NOT colour(n)(cs))
      IMPLIES
    blacks(N1, N2)(set_colour(n, TRUE)(cs)) = blacks(N1, N2)(cs) + 1

  blacks5 : LEMMA
    (blacks(0, NODES)(set_colour(n, TRUE)(cs)) = blacks(0, NODES)(cs)) IMPLIES colour(n)(cs)

  blacks6 : LEMMA
    NOT colour(n2)(cs) IMPLIES blacks(n1, n2 + 1)(cs) = blacks(n1, n2)(cs)

  blacks7 : LEMMA
    n1 <= n2 AND colour(n2)(cs) IMPLIES blacks(n1, n2 + 1)(cs) = blacks(n1, n2)(cs) + 1

END Memory_Properties
```

A.3 The Refinement Steps

A.3.1 Top Level Specification

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Garbage_Collector :                                      %
%   The top-level specification of the garbage collector. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Garbage_Collector[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Memory[NODES,SONS,ROOTS]

  State   : TYPE = Memory
  O_State : TYPE = Memory

  s,s1,s2 : VAR State
  n,k     : VAR Node
  i       : VAR Index

  proj(s):O_State = s

  init(s):bool = (s = null_array)

  Rule_mutate(n,i,k)(s):State =
    IF accessible(k)(s) THEN
      set_son(n,i,k)(s)
    ELSE
      s
    ENDIF

  Rule_append(n)(s):State =
    IF NOT accessible(n)(s) THEN
      append_to_free(n)(s)
    ELSE
      s
    ENDIF

  next(s1,s2):bool =
```

```
    (EXISTS n,i,k: s2 = Rule_mutate(n,i,k)(s1)) OR
    (EXISTS n: s2 = Rule_append(n)(s1)) OR
    s2 = s1

END Garbage_Collector
```

A.3.2 First Refinement

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Garbage_Collector1 :                             %
%   The first refinement of the garbage collector. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Garbage_Collector1[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Coloured_Memory[NODES,SONS,ROOTS]

  MuPC    : TYPE = {MUTATE,COLOUR}
  CoPC    : TYPE = {COLOUR,TEST_L,APPEND}

  State   : TYPE = [# MU : MuPC, CHI : CoPC,
                      Q : nat, L : nat , C : Colours, M : Memory #]

  O_State : TYPE = Memory

  s,s1,s2 : VAR State
  n,k     : VAR Node
  i       : VAR Index

  proj(s):O_State = M(s)

  init(s):bool =
      MU(s) = MUTATE
    & CHI(s) = COLOUR
    & M(s) = null_array


  %%%%%%%%%%%%%%%%%%%%%%%
  % The MUTATOR Process %
  %%%%%%%%%%%%%%%%%%%%%%%

  Rule_mutate(n,i,k)(s):State =
    IF MU(s) = MUTATE AND accessible(k)(M(s)) THEN
      s WITH [M := set_son(n,i,k)(M(s)),
              Q := k,
```

```
            MU := COLOUR]
  ELSE
    s
  ENDIF

Rule_colour_target(s):State =
  IF MU(s) = COLOUR AND Q(s) < NODES THEN
    s WITH [C := set_colour(Q(s),TRUE)(C(s)),
            MU := MUTATE]
  ELSE
    s
  ENDIF

MUTATOR(s1,s2):bool =
     (EXISTS n,i,k: s2 = Rule_mutate(n,i,k)(s1))
  OR s2 = Rule_colour_target(s1)


%%%%%%%%%%%%%%%%%%%%%%%%%
% The COLLECTOR Process %
%%%%%%%%%%%%%%%%%%%%%%%%%

Rule_stop_colouring(s):State =
  IF CHI(s) = COLOUR AND blackened(C(s),M(s)) THEN
    s WITH [L := 0, CHI := TEST_L]
  ELSE
    s
  ENDIF

Rule_colour(n)(s):State =
  IF CHI(s) = COLOUR THEN
    s WITH [C := set_colour(n,TRUE)(C(s))]
  ELSE
    s
  ENDIF

Rule_stop_appending(s):State =
  IF CHI(s) = TEST_L AND L(s) = NODES THEN
    s WITH [CHI := COLOUR]
  ELSE
    s
  ENDIF

Rule_continue_appending(s):State =
  IF CHI(s) = TEST_L AND L(s) < NODES THEN
    s WITH [CHI := APPEND]
  ELSE
```

```
      s
    ENDIF

  Rule_black_to_white(s):State =
    IF CHI(s) = APPEND AND L(s) < NODES AND colour(L(s))(C(s)) THEN
      s WITH [C := set_colour(L(s),FALSE)(C(s)),
              L := L(s)+1,
              CHI := TEST_L]
    ELSE
      s
    ENDIF

  Rule_append_white(s):State =
    IF CHI(s) = APPEND AND L(s) < NODES AND NOT colour(L(s))(C(s)) THEN
      s WITH [M := append_to_free(L(s))(M(s)),
              L := L(s) + 1,
              CHI := TEST_L]
    ELSE
      s
    ENDIF

  COLLECTOR(s1,s2):bool =
       s2 = Rule_stop_colouring(s1)
    OR (EXISTS n: s2 = Rule_colour(n)(s1))
    OR s2 = Rule_stop_appending(s1)
    OR s2 = Rule_continue_appending(s1)
    OR s2 = Rule_black_to_white(s1)
    OR s2 = Rule_append_white(s1)

  %%%%%%%%%%%%%%%%%%%%%%%%%%%
  % The Transition Relation %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%

  next(s1,s2):bool =
    MUTATOR(s1,s2) OR
    COLLECTOR(s1,s2) OR
    s2 = s1

END Garbage_Collector1
```

A.3.3 Second Refinement

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Garbage_Collector2 :                              %
%   The second refinement of the garbage collector. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Garbage_Collector2[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Coloured_Memory[NODES,SONS,ROOTS]

  MuPC    : TYPE = {MUTATE,COLOUR}
  CoPC    : TYPE = {COLOUR_ROOTS,PROPAGATE,TEST_L,APPEND}

  State   : TYPE = [# MU : MuPC, CHI : CoPC,
                    Q : nat, K : nat, L : nat, C : Colours, M : Memory #]

  O_State : TYPE = Memory

  s,s1,s2 : VAR State
  n,k     : VAR Node
  i       : VAR Index

  proj(s):O_State = M(s)

  init(s):bool =
      MU(s) = MUTATE
    & CHI(s) = COLOUR_ROOTS
    & K(s) = 0
    & M(s) = null_array

  %%%%%%%%%%%%%%%%%%%%%%%
  % The MUTATOR Process %
  %%%%%%%%%%%%%%%%%%%%%%%

  Rule_mutate(n,i,k)(s):State =
    IF MU(s) = MUTATE AND accessible(k)(M(s)) THEN
      s WITH [M := set_son(n,i,k)(M(s)),
              Q := k,
```

```
            MU := COLOUR]
  ELSE
    s
  ENDIF

Rule_colour_target(s):State =
  IF MU(s) = COLOUR AND Q(s) < NODES THEN
    s WITH [C := set_colour(Q(s),TRUE)(C(s)),
            MU := MUTATE]
  ELSE
    s
  ENDIF

MUTATOR(s1,s2):bool =
     (EXISTS n,i,k: s2 = Rule_mutate(n,i,k)(s1))
  OR s2 = Rule_colour_target(s1)


%%%%%%%%%%%%%%%%%%%%%%%%%
% The COLLECTOR Process %
%%%%%%%%%%%%%%%%%%%%%%%%%

Rule_stop_colouring_roots(s):State =
  IF CHI(s) = COLOUR_ROOTS AND K(s) = ROOTS THEN
    s WITH [CHI := PROPAGATE]
  ELSE
    s
  ENDIF

Rule_colour_root(s):State =
  IF CHI(s) = COLOUR_ROOTS AND K(s) < ROOTS THEN
    s WITH [C := set_colour(K(s),TRUE)(C(s)),
            K := K(s) + 1]
  ELSE
    s
  ENDIF

Rule_stop_propagating(s):State =
  IF CHI(s) = PROPAGATE AND NOT exists_bw(C(s),M(s)) THEN
    s WITH [L := 0, CHI := TEST_L]
  ELSE
    s
  ENDIF

Rule_propagate(n,i)(s):State =
  IF CHI(s) = PROPAGATE AND bw(n,i)(C(s),M(s)) THEN
    s WITH [C := set_colour(son(n,i)(M(s)),TRUE)(C(s))]
```

```
  ELSE
    s
  ENDIF

Rule_stop_appending(s):State =
  IF CHI(s) = TEST_L AND L(s) = NODES THEN
    s WITH [K := 0, CHI := COLOUR_ROOTS]
  ELSE
    s
  ENDIF

Rule_continue_appending(s):State =
  IF CHI(s) = TEST_L AND L(s) < NODES THEN
    s WITH [CHI := APPEND]
  ELSE
    s
  ENDIF

Rule_black_to_white(s):State =
  IF CHI(s) = APPEND AND L(s) < NODES AND colour(L(s))(C(s)) THEN
    s WITH [C := set_colour(L(s),FALSE)(C(s)),
            L := L(s)+1,
            CHI := TEST_L]
  ELSE
    s
  ENDIF

Rule_append_white(s):State =
  IF CHI(s) = APPEND AND L(s) < NODES AND NOT colour(L(s))(C(s)) THEN
    s WITH [M := append_to_free(L(s))(M(s)),
            L := L(s) + 1,
            CHI := TEST_L]
  ELSE
    s
  ENDIF

COLLECTOR(s1,s2):bool =
     s2 = Rule_stop_colouring_roots(s1)
  OR s2 = Rule_colour_root(s1)
  OR s2 = Rule_stop_propagating(s1)
  OR (EXISTS n,i: s2 = Rule_propagate(n,i)(s1))
  OR s2 = Rule_stop_appending(s1)
  OR s2 = Rule_continue_appending(s1)
  OR s2 = Rule_black_to_white(s1)
  OR s2 = Rule_append_white(s1)

%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
  % The Transition Relation %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%

  next(s1,s2):bool =
    MUTATOR(s1,s2) OR
    COLLECTOR(s1,s2) OR
    s2 = s1

END Garbage_Collector2
```

A.3.4 Third Refinement

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Garbage_Collector3 :                            %
%   The third refinement of the garbage collector. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Garbage_Collector3[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Coloured_Memory[NODES,SONS,ROOTS]

  MuPC    : TYPE = {MUTATE,COLOUR}
  CoPC    : TYPE = {COLOUR_ROOTS,TEST_I,TEST_COLOUR,COLOUR_SONS,
                    TEST_H,COUNT,COMPARE,TEST_L,APPEND}

  State   : TYPE = [# MU : MuPC, CHI : CoPC,
                      Q : nat, BC : nat, OBC : nat,
                      H : nat, I : nat, J : nat, K : nat, L : nat,
                      C : Colours, M : Memory #]

  O_State : TYPE = Memory

  s,s1,s2 : VAR State
  n,k     : VAR Node
  i       : VAR Index

  proj(s):O_State = M(s)

  init(s):bool =
      MU(s) = MUTATE
    & CHI(s) = COLOUR_ROOTS
    & OBC(s) = 0
    & K(s) = 0
    & M(s) = null_array

  %%%%%%%%%%%%%%%%%%%%%%%
  % The MUTATOR Process %
  %%%%%%%%%%%%%%%%%%%%%%%
```

```
Rule_mutate(n,i,k)(s):State =
  IF MU(s) = MUTATE AND accessible(k)(M(s)) THEN
    s WITH [M := set_son(n,i,k)(M(s)),
            Q := k,
            MU := COLOUR]
  ELSE
    s
  ENDIF

Rule_colour_target(s):State =
  IF MU(s) = COLOUR AND Q(s) < NODES THEN
    s WITH [C := set_colour(Q(s),TRUE)(C(s)),
            MU := MUTATE]
  ELSE
    s
  ENDIF

MUTATOR(s1,s2):bool =
     (EXISTS n,i,k: s2 = Rule_mutate(n,i,k)(s1))
  OR s2 = Rule_colour_target(s1)


%%%%%%%%%%%%%%%%%%%%%%%%%
% The COLLECTOR Process %
%%%%%%%%%%%%%%%%%%%%%%%%%

Rule_stop_colouring_roots(s):State =
  IF CHI(s) = COLOUR_ROOTS AND K(s) = ROOTS THEN
    s WITH [I := 0, CHI := TEST_I]
  ELSE
    s
  ENDIF

Rule_colour_root(s):State =
  IF CHI(s) = COLOUR_ROOTS AND K(s) < ROOTS THEN
    s WITH [C := set_colour(K(s),TRUE)(C(s)),
            K := K(s) + 1]
  ELSE
    s
  ENDIF

Rule_stop_propagating(s):State =
  IF CHI(s) = TEST_I AND I(s) = NODES THEN
    s WITH [BC := 0, H := 0, CHI := TEST_H]
  ELSE
    s
  ENDIF
```

```
Rule_continue_propagating(s):State =
  IF CHI(s) = TEST_I AND I(s) < NODES THEN
    s WITH [CHI := TEST_COLOUR]
  ELSE
    s
  ENDIF

Rule_white_node(s):State =
  IF CHI(s) = TEST_COLOUR AND I(s) < NODES AND NOT colour(I(s))(C(s)) THEN
    s WITH [I := I(s) + 1, CHI := TEST_I]
  ELSE
    s
  ENDIF

Rule_black_node(s):State =
  IF CHI(s) = TEST_COLOUR AND I(s) < NODES AND colour(I(s))(C(s)) THEN
    s WITH [J := 0, CHI := COLOUR_SONS]
  ELSE
    s
  ENDIF

Rule_stop_colouring_sons(s):State =
  IF CHI(s) = COLOUR_SONS AND J(s) = SONS THEN
    s WITH [I := I(s) + 1, CHI := TEST_I]
  ELSE
    s
  ENDIF

Rule_colour_son(s):State =
  IF CHI(s) = COLOUR_SONS AND I(s) < NODES AND J(s) < SONS THEN
    s WITH [C := set_colour(son(I(s),J(s))(M(s)),TRUE)(C(s)),
            J := J(s) + 1]
  ELSE
    s
  ENDIF

Rule_stop_counting(s):State =
  IF CHI(s) = TEST_H AND H(s) = NODES THEN
    s WITH [CHI := COMPARE]
  ELSE
    s
  ENDIF

Rule_continue_counting(s):State =
  IF CHI(s) = TEST_H AND H(s) /= NODES THEN
    s WITH [CHI := COUNT]
```

```
  ELSE
    s
  ENDIF

Rule_skip_white(s):State =
  IF CHI(s) = COUNT AND H(s) < NODES AND NOT colour(H(s))(C(s)) THEN
    s WITH [H := H(s) + 1, CHI := TEST_H]
  ELSE
    s
  ENDIF

Rule_count_black(s):State =
  IF CHI(s) = COUNT AND H(s) < NODES AND colour(H(s))(C(s)) THEN
    s WITH [BC := BC(s) + 1, H := H(s) + 1, CHI := TEST_H]
  ELSE
    s
  ENDIF

Rule_stop_colouring(s):State =
  IF CHI(s) = COMPARE AND BC(s) = OBC(s) THEN
    s WITH [L := 0, CHI := TEST_L]
  ELSE
    s
  ENDIF

Rule_continue_colouring(s):State =
  IF CHI(s) = COMPARE AND BC(s) /= OBC(s) THEN
    s WITH [OBC := BC(s), I := 0, CHI := TEST_I]
  ELSE
    s
  ENDIF

Rule_stop_appending(s):State =
  IF CHI(s) = TEST_L AND L(s) = NODES THEN
    s WITH [BC := 0, OBC := 0, K := 0, CHI := COLOUR_ROOTS]
  ELSE
    s
  ENDIF

Rule_continue_appending(s):State =
  IF CHI(s) = TEST_L AND L(s) < NODES THEN
    s WITH [CHI := APPEND]
  ELSE
    s
  ENDIF

Rule_black_to_white(s):State =
```

```
    IF CHI(s) = APPEND AND L(s) < NODES AND colour(L(s))(C(s)) THEN
      s WITH [C := set_colour(L(s),FALSE)(C(s)),
              L := L(s)+1, CHI := TEST_L]
    ELSE
      s
    ENDIF

  Rule_append_white(s):State =
    IF CHI(s) = APPEND AND L(s) < NODES AND NOT colour(L(s))(C(s)) THEN
      s WITH [M := append_to_free(L(s))(M(s)),
              L := L(s) + 1, CHI := TEST_L]
    ELSE
      s
    ENDIF

  COLLECTOR(s1,s2):bool =
     s2 = Rule_stop_colouring_roots(s1)
  OR s2 = Rule_colour_root(s1)
  OR s2 = Rule_stop_propagating(s1)
  OR s2 = Rule_continue_propagating(s1)
  OR s2 = Rule_white_node(s1)
  OR s2 = Rule_black_node(s1)
  OR s2 = Rule_stop_colouring_sons(s1)
  OR s2 = Rule_colour_son(s1)
  OR s2 = Rule_stop_counting(s1)
  OR s2 = Rule_continue_counting(s1)
  OR s2 = Rule_skip_white(s1)
  OR s2 = Rule_count_black(s1)
  OR s2 = Rule_stop_colouring(s1)
  OR s2 = Rule_continue_colouring(s1)
  OR s2 = Rule_stop_appending(s1)
  OR s2 = Rule_continue_appending(s1)
  OR s2 = Rule_black_to_white(s1)
  OR s2 = Rule_append_white(s1)


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % The Transition Relation %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%

  next(s1,s2):bool =
    MUTATOR(s1,s2) OR
    COLLECTOR(s1,s2) OR
    s2 = s1

END Garbage_Collector3
```

A.4 Refinement and Invariant Lemmas

A.4.1 Lemmas in First Refinement

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Refinement1 :                                               %
%   Applies the ``Refinement'' theory to yield                %
%   the first refinement lemma ``R1''. There is a             %
%   ``sim_xxx'' lemma for each concrete transition ``xxx'',   %
%   and these lemmas are used in proving the ``next_h'' lemma, %
%   which again is used in proving the TCC's generated by the  %
%   application of the ``Refinement'' theory.                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Refinement1[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  S  : THEORY = Garbage_Collector [NODES,SONS,ROOTS]
  I1 : THEORY = Garbage_Collector1[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector1_Inv[NODES,SONS,ROOTS]

  s      : VAR I1.State
  r1,r2 : VAR (I)
  n,k    : VAR Node
  i      : VAR Index
  cs     : VAR Colours

  abs(s):S.State = M(s)

  sim_mutate : LEMMA
    (EXISTS n,i,k: r2 = Rule_mutate(n,i,k)(r1)) IMPLIES
      (EXISTS n,i,k: abs(r2) = Rule_mutate(n,i,k)(abs(r1))) OR
      abs(r2) = abs(r1)

  sim_colour_target : LEMMA
    r2 = Rule_colour_target(r1) IMPLIES
      abs(r2) = abs(r1)

  sim_stop_colouring : LEMMA
    r2 = Rule_stop_colouring(r1) IMPLIES
```

```
      abs(r2) = abs(r1)

  sim_colour : LEMMA
    (EXISTS n: r2 = Rule_colour(n)(r1)) IMPLIES
      abs(r2) = abs(r1)

  sim_stop_appending : LEMMA
    r2 = Rule_stop_appending(r1) IMPLIES
      abs(r2) = abs(r1)

  sim_continue_appending : LEMMA
    r2 = Rule_continue_appending(r1) IMPLIES
      abs(r2) = abs(r1)

  sim_black_to_white : LEMMA
    r2 = Rule_black_to_white(r1) IMPLIES
      abs(r2) = abs(r1)

  sim_append_white : LEMMA
    r2 = Rule_append_white(r1) IMPLIES
      (EXISTS n: abs(r2) = Rule_append(n)(abs(r1))) OR abs(r2) = abs(r1)

  next_h : LEMMA
    next(r1,r2) IMPLIES next(abs(r1),abs(r2))

  R1 : THEORY =
    Refinement[S.O_State,
      S.State,S.init,S.next,S.proj,
      I1.State,I1.init,I1.next,I1.proj,
      abs,I]

END Refinement1
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Garbage_Collector1_Inv :                                       %
%   Defines all invariants used in proving the first refinement. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Garbage_Collector1_Inv[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Memory_Properties[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector1[NODES,SONS,ROOTS]
  IMPORTING Invariant_Predicates[State]

  s : VAR State

  inv1(s):bool =
    CHI(s)=APPEND AND L(s) < NODES AND accessible(L(s))(M(s))
      IMPLIES
    colour(L(s))(C(s))

  inv2(s):bool =
    CHI(s)=TEST_L OR CHI(s)=APPEND IMPLIES blackened(L(s))(C(s),M(s))

  I : pred[State] = inv1 & inv2

  pi : pred[pred[State]] = preserved(init,next)(I)

  i_inv1 : LEMMA I IMPLIES inv1
  i_inv2 : LEMMA I IMPLIES inv2

  p_inv1 : LEMMA pi(inv1)
  p_inv2 : LEMMA pi(inv2)

  p_I : LEMMA pi(I)

  inv : LEMMA invariant(init,next)(I)

END Garbage_Collector1_Inv
```

A.4.2 Lemmas in Second Refinement

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Refinement2 :                               %
%   Applies the ``Refinement'' theory to yield %
%   the second refinement lemma ``R2''.       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Refinement2[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  I1 : THEORY = Garbage_Collector1[NODES,SONS,ROOTS]
  I2 : THEORY = Garbage_Collector2[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector2_Inv[NODES,SONS,ROOTS]

  s     : VAR I2.State
  r1,r2 : VAR (I)
  n,k   : VAR Node
  i     : VAR Index
  cs    : VAR Colours

  abs(s):I1.State =
    (# MU := CASES MU(s) OF
               MUTATE : MUTATE,
               COLOUR : COLOUR
             ENDCASES,
       CHI := CASES CHI(s) OF
                COLOUR_ROOTS : COLOUR,
                   PROPAGATE : COLOUR,
                      TEST_L : TEST_L,
                      APPEND : APPEND
              ENDCASES,
       Q   := Q(s),
       L   := L(s),
       C   := C(s),
       M   := M(s)
     #)

  sim_mutate : LEMMA
    (EXISTS n,i,k: r2 = Rule_mutate(n,i,k)(r1)) IMPLIES
```

```
    EXISTS n,i,k: abs(r2) = Rule_mutate(n,i,k)(abs(r1))

sim_colour_target : LEMMA
  r2 = Rule_colour_target(r1) IMPLIES
    abs(r2) = Rule_colour_target(abs(r1))

sim_stop_colouring_roots : LEMMA
  r2 = Rule_stop_colouring_roots(r1) IMPLIES
    abs(r2) = abs(r1)

sim_colour_root : LEMMA
  r2 = Rule_colour_root(r1) IMPLIES
    (EXISTS n: abs(r2) = Rule_colour(n)(abs(r1))) OR abs(r2) = abs(r1)

sim_stop_propagating : LEMMA
  r2 = Rule_stop_propagating(r1) IMPLIES
    abs(r2) = Rule_stop_colouring(abs(r1)) OR abs(r2) = abs(r1)

sim_propagate : LEMMA
  (EXISTS n,i: r2 = Rule_propagate(n,i)(r1)) IMPLIES
    (EXISTS k: abs(r2) = Rule_colour(k)(abs(r1))) OR
    abs(r2) = abs(r1)

sim_stop_appending : LEMMA
  r2 = Rule_stop_appending(r1) IMPLIES
    abs(r2) = Rule_stop_appending(abs(r1))

sim_continue_appending : LEMMA
  r2 = Rule_continue_appending(r1) IMPLIES
    abs(r2) = Rule_continue_appending(abs(r1))

sim_black_to_white : LEMMA
  r2 = Rule_black_to_white(r1) IMPLIES
    abs(r2) = Rule_black_to_white(abs(r1))

sim_append_white : LEMMA
  r2 = Rule_append_white(r1) IMPLIES
    abs(r2) = Rule_append_white(abs(r1))

next_h : LEMMA
  next(r1,r2) IMPLIES next(abs(r1),abs(r2))

R2 : THEORY =
  Refinement[I1.O_State,
    I1.State,I1.init,I1.next,I1.proj,
    I2.State,I2.init,I2.next,I2.proj,
    abs,I]
```

```
END Refinement2
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Garbage_Collector2_Inv :                                        %
%   Defines all invariants used in proving the second refinement. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Garbage_Collector2_Inv[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Memory_Properties[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector2[NODES,SONS,ROOTS]
  IMPORTING Invariant_Predicates[State]

  s : VAR State

  inv1(s):bool =
    CHI(s)=PROPAGATE IMPLIES black_roots(ROOTS)(C(s))

  inv2(s):bool =
    CHI(s)=COLOUR_ROOTS IMPLIES black_roots(K(s))(C(s))

  I : pred[State] = inv1 & inv2

  pi : pred[pred[State]] = preserved(init,next)(I)

  i_inv1 : LEMMA I IMPLIES inv1
  i_inv2 : LEMMA I IMPLIES inv2

  p_inv1 : LEMMA pi(inv1)
  p_inv2 : LEMMA pi(inv2)

  p_I : LEMMA pi(I)

  inv : LEMMA invariant(init,next)(I)

END Garbage_Collector2_Inv
```

A.4.3 Lemmas in Third Refinement

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Refinement3 :                               %
%   Applies the ``Refinement'' theory to yield %
%   the third refinement lemma ``R3''.         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Refinement3[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  I2 : THEORY = Garbage_Collector2[NODES,SONS,ROOTS]
  I3 : THEORY = Garbage_Collector3[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector3_Inv[NODES,SONS,ROOTS]

  s     : VAR I3.State
  r1,r2 : VAR (I)
  n,k   : VAR Node
  i     : VAR Index
  cs    : VAR Colours

  abs(s):I2.State =
    (# MU := CASES MU(s) OF
               MUTATE : MUTATE,
               COLOUR : COLOUR
             ENDCASES,
       CHI := CASES CHI(s) OF
                COLOUR_ROOTS : COLOUR_ROOTS,
                      TEST_I : PROPAGATE,
                 TEST_COLOUR : PROPAGATE,
                 COLOUR_SONS : PROPAGATE,
                      TEST_H : PROPAGATE,
                       COUNT : PROPAGATE,
                     COMPARE : PROPAGATE,
                      TEST_L : TEST_L,
                      APPEND : APPEND
              ENDCASES,
       Q := Q(s),
       K := K(s),
       L := L(s),
```

```
    C := C(s),
    M := M(s)
  #)

sim_mutate : LEMMA
  (EXISTS n,i,k: r2 = Rule_mutate(n,i,k)(r1)) IMPLIES
    EXISTS n,i,k: abs(r2) = Rule_mutate(n,i,k)(abs(r1))

sim_colour_target : LEMMA
  r2 = Rule_colour_target(r1) IMPLIES
    abs(r2) = Rule_colour_target(abs(r1))

sim_stop_colouring_roots : LEMMA
  r2 = Rule_stop_colouring_roots(r1) IMPLIES
    abs(r2) = Rule_stop_colouring_roots(abs(r1))

sim_colour_root : LEMMA
  r2 = Rule_colour_root(r1) IMPLIES
    abs(r2) = Rule_colour_root(abs(r1))

sim_stop_propagating : LEMMA
  r2 = Rule_stop_propagating(r1) IMPLIES
    abs(r2) = abs(r1)

sim_continue_propagating : LEMMA
  r2 = Rule_continue_propagating(r1) IMPLIES
    abs(r2) = abs(r1)

sim_white_node : LEMMA
  r2 = Rule_white_node(r1) IMPLIES
    abs(r2) = abs(r1)

sim_black_node : LEMMA
  r2 = Rule_black_node(r1) IMPLIES
    abs(r2) = abs(r1)

sim_stop_colouring_sons : LEMMA
  r2 = Rule_stop_colouring_sons(r1) IMPLIES
    abs(r2) = abs(r1)

sim_colour_son : LEMMA
  r2 = Rule_colour_son(r1) IMPLIES
    (EXISTS n,i: abs(r2) = Rule_propagate(n,i)(abs(r1))) OR
    abs(r2) = abs(r1)

sim_stop_counting : LEMMA
  r2 = Rule_stop_counting(r1) IMPLIES
```

```
    abs(r2) = abs(r1)

sim_continue_counting : LEMMA
  r2 = Rule_continue_counting(r1) IMPLIES
    abs(r2) = abs(r1)

sim_skip_white : LEMMA
  r2 = Rule_skip_white(r1) IMPLIES
    abs(r2) = abs(r1)

sim_count_black : LEMMA
  r2 = Rule_count_black(r1) IMPLIES
    abs(r2) = abs(r1)

sim_stop_colouring : LEMMA
  r2 = Rule_stop_colouring(r1) IMPLIES
    abs(r2) = Rule_stop_propagating(abs(r1)) OR
    abs(r2) = abs(r1)

sim_continue_colouring : LEMMA
  r2 = Rule_continue_colouring(r1) IMPLIES
    abs(r2) = abs(r1)

sim_stop_appending : LEMMA
  r2 = Rule_stop_appending(r1) IMPLIES
    abs(r2) = Rule_stop_appending(abs(r1))

sim_continue_appending : LEMMA
  r2 = Rule_continue_appending(r1) IMPLIES
    abs(r2) = Rule_continue_appending(abs(r1))

sim_black_to_white : LEMMA
  r2 = Rule_black_to_white(r1) IMPLIES
    abs(r2) = Rule_black_to_white(abs(r1))

sim_append_white : LEMMA
  r2 = Rule_append_white(r1) IMPLIES
    abs(r2) = Rule_append_white(abs(r1))

next_h : LEMMA
  next(r1,r2) IMPLIES next(abs(r1),abs(r2))

R3 : THEORY =
  Refinement[I2.O_State,
    I2.State,I2.init,I2.next,I2.proj,
    I3.State,I3.init,I3.next,I3.proj,
    abs,I]
```

```
END Refinement3
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Garbage_Collector3_Inv :                                      %
%   Defines all invariants used in proving the third refinement. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Garbage_Collector3_Inv[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Memory_Properties[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector3[NODES,SONS,ROOTS]
  IMPORTING Invariant_Predicates[State]

  s : VAR State

  inv1(s):bool =
    CHI(s)=COLOUR_SONS AND I(s) < NODES IMPLIES colour(I(s))(C(s))

  inv2(s):bool =
    CHI(s)=COMPARE AND BC(s)=OBC(s) IMPLIES NOT exists_bw(C(s),M(s))

  inv3(s):bool =
    (CHI(s)=COLOUR_ROOTS OR
     CHI(s)=TEST_I OR CHI(s)=TEST_COLOUR OR CHI(s)=COLOUR_SONS OR
     CHI(s)=TEST_H OR CHI(s)=COUNT OR CHI(s)=COMPARE)
       IMPLIES
     black_roots(IF CHI(s)=COLOUR_ROOTS THEN K(s) ELSE ROOTS ENDIF)(C(s))

  inv4(s):bool =
    MU(s)=COLOUR AND Q(s) < NODES IMPLIES accessible(Q(s))(M(s))

  inv5(s):bool =
    (CHI(s)=TEST_H OR CHI(s)=COUNT OR CHI(s)=COMPARE) AND
    OBC(s) = BC(s) + blacks(H(s),NODES)(C(s))
      IMPLIES
    NOT exists_bw(C(s),M(s))

  inv6(s):bool =
    CHI(s)=APPEND AND L(s) < NODES AND accessible(L(s))(M(s))
      IMPLIES
```

```
  colour(L(s))(C(s))

inv7(s):bool =
  ((CHI(s)=TEST_I OR CHI(s)=TEST_COLOUR OR CHI(s)=COLOUR_SONS) AND
    OBC(s) = blacks(0,NODES)(C(s)) AND
    exists_bw(0,0,I(s),IF CHI(s)=COLOUR_SONS THEN J(s) ELSE 0 ENDIF)(C(s),M(s)))
  IMPLIES
    exists_bw(I(s),IF CHI(s)=COLOUR_SONS THEN J(s) ELSE 0 ENDIF,NODES,0)
      (C(s),M(s))

inv8(s):bool =
  (CHI(s)=TEST_H OR CHI(s)=COUNT OR CHI(s)=COMPARE)
    IMPLIES
  OBC(s) <= BC(s) + blacks(H(s),NODES)(C(s))

inv9(s):bool =
  ((CHI(s)=TEST_I OR CHI(s)=TEST_COLOUR OR CHI(s)=COLOUR_SONS) AND
    OBC(s) = blacks(0,NODES)(C(s)) AND
    exists_bw(0,0,I(s),IF CHI(s)=COLOUR_SONS THEN J(s) ELSE 0 ENDIF)(C(s),M(s)))
  IMPLIES
    MU(s)=COLOUR

inv10(s):bool =
  (CHI(s)=COLOUR_ROOTS OR
   CHI(s)=TEST_I OR CHI(s)=TEST_COLOUR OR CHI(s)=COLOUR_SONS)
    IMPLIES
  OBC(s) <= blacks(0,NODES)(C(s))

inv11(s):bool =
  FORALL (n:Node,i:Index):
    (((CHI(s)=TEST_I OR CHI(s)=TEST_COLOUR OR CHI(s)=COLOUR_SONS) AND
        OBC(s) = blacks(0,NODES)(C(s)) AND
       (n,i) < (I(s),IF CHI(s)=COLOUR_SONS THEN J(s) ELSE 0 ENDIF) AND
       bw(n,i)(C(s),M(s)))
    IMPLIES
      (MU(s)=COLOUR AND son(n,i)(M(s))=Q(s)))

inv12(s):bool =
  ((CHI(s)=TEST_COLOUR OR CHI(s)=COLOUR_SONS) IMPLIES I(s) < NODES)

inv13(s):bool =
  (CHI(s)=TEST_L OR CHI(s)=APPEND)
    IMPLIES
  blackened(L(s))(C(s),M(s))

inv14(s):bool =
  CHI(s)=COMPARE IMPLIES BC(s) <= blacks(0,NODES)(C(s))
```

```
inv15(s):bool =
  (CHI(s)=TEST_H OR CHI(s)=COUNT) IMPLIES BC(s) <= blacks(0,H(s))(C(s))

I : pred[State] =
  inv1 & inv2  & inv3  & inv4  & inv5  & inv6  & inv7 &
  inv8 & inv10 & inv11 & inv12 & inv13 & inv14 & inv15

pi : pred[pred[State]] = preserved(init,next)(I)

c_inv9 : LEMMA inv11 IMPLIES inv9

i_inv1  : LEMMA I IMPLIES inv1
i_inv2  : LEMMA I IMPLIES inv2
i_inv3  : LEMMA I IMPLIES inv3
i_inv4  : LEMMA I IMPLIES inv4
i_inv5  : LEMMA I IMPLIES inv5
i_inv6  : LEMMA I IMPLIES inv6
i_inv7  : LEMMA I IMPLIES inv7
i_inv8  : LEMMA I IMPLIES inv8
i_inv9  : LEMMA I IMPLIES inv9
i_inv10 : LEMMA I IMPLIES inv10
i_inv11 : LEMMA I IMPLIES inv11
i_inv12 : LEMMA I IMPLIES inv12
i_inv13 : LEMMA I IMPLIES inv13
i_inv14 : LEMMA I IMPLIES inv14
i_inv15 : LEMMA I IMPLIES inv15

p_inv1  : LEMMA pi(inv1)
p_inv2  : LEMMA pi(inv2)
p_inv3  : LEMMA pi(inv3)
p_inv4  : LEMMA pi(inv4)
p_inv5  : LEMMA pi(inv5)
p_inv6  : LEMMA pi(inv6)
p_inv7  : LEMMA pi(inv7)
p_inv8  : LEMMA pi(inv8)
p_inv10 : LEMMA pi(inv10)
p_inv11 : LEMMA pi(inv11)
p_inv12 : LEMMA pi(inv12)
p_inv13 : LEMMA pi(inv13)
p_inv14 : LEMMA pi(inv14)
p_inv15 : LEMMA pi(inv15)

p_I : LEMMA pi(I)

inv : LEMMA invariant(init,next)(I)
```

```
END Garbage_Collector3_Inv
```

A.4.4 Final Refinement Theorem

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Composed_Refinement :                                            %
%   States the final correctness criteria in theorem ``ref'',      %
%   which says that the third implementation refines the top-level %
%   specification. The proof uses the transitivity property of the %
%   refinement relation.                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Composed_Refinement[
  NODES : posnat,
  SONS  : posnat,
  ROOTS : posnat] : THEORY

BEGIN

  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING

  IMPORTING Refinement1[NODES,SONS,ROOTS]
  IMPORTING Refinement2[NODES,SONS,ROOTS]
  IMPORTING Refinement3[NODES,SONS,ROOTS]

  IMPORTING Refine_Predicate
  IMPORTING Refine_Predicate_Transitive

  ref2 : LEMMA
    refines[S.O_State,S.State,I2.State]
      (I2.init,I2.next,I2.proj)(S.init,S.next,S.proj)

  ref : THEOREM
    refines[S.O_State,S.State,I3.State]
      (I3.init,I3.next,I3.proj)(S.init,S.next,S.proj)

END Composed_Refinement
```

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[2] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Toplas*, 6, July 1984.

[3] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *Computer Aided Verification 96*, Lecture Notes in Computer Science, 1996. To appear.

[4] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[5] J. L. A. Van de Snepscheut. "algorithms for on-the-fly garbage collection" revisited. *Information Processing Letters*, 24, March 1987.

[6] E. W. Dijkstra, L. Lamport, A.J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *ACM*, 21, November 1978.

[7] K. Havelund. Mechanical verification of a garbage collector. May 1996.

[8] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer-Verlag, 1996.

[9] L. Lamport. The Temporal Logic of Actions. Technical report, Digital Equipment Corporation (DEC) Systems Research Center, Palo Alto, California, USA, April 1994.

[10] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. 21(2):107–125, February 1995.

[11] C. Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3, 1988.

[12] David M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[13] N. Shankar. Mechanized verification of real-time systems using PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, USA, March 1993.