Verifying a Garbage Collection Algorithm*†‡

Paul B. Jackson

Department of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, UK
pbj@dcs.ed.ac.uk
http://www.dcs.ed.ac.uk/home/pbj

June 8, 1998

Abstract

We present a case study in using the PVS interactive theorem prover to formally model and verify properties of a tricolour garbage collection algorithm. We model the algorithm using state transition systems and verify safety and liveness properties in linear temporal logic. We set up two systems, each of which models the algorithm itself, object allocation, and the behaviour of user programs. The models differ in how concretely they model the heap. We verify the properties of the more abstract system, and then, once a refinement relation is exhibited between the systems, we show the more concrete system to have corresponding properties.

We discuss the linear temporal logic framework we set up, commenting in particular on how we handle fairness and how we use a 'leads-to-via' predicate to reason about the propagation of properties that are stable in specified regions of system state spaces. We also describe strategies (tactics) we wrote to improve the quality of interaction and increase the degree of automation.

1 Introduction

This case study is part of larger project at the University of Edinburgh to develop and assess formal models and verification techniques for garbage collection algorithms. This project is being carried out in consultation with the memory management group of the software house Harlequin[1] which, amongst other things,

†To appear in: Jim Grundy and Malcolm Newey editors, proceedings of *11th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'98*. Lecture Notes in Computer Science, Springer-Verlag. September 1998. From September onwards, full bibliographic details should be available from my homepage.

‡This work was supported by the UK Engineering and Physical Sciences Research Council under grant GR/J85509 and, while the author was a Visiting Fellow in the Computer Science Laboratory of SRI International in Menlo Park, California USA, by the US National Science Foundation under contract CCR-9509931.

[1] http://www.harlequin.com/

produces compilers for Lisp, ML and Dylan. One of the primary goals of our project is to demonstrate that formal techniques can have a positive impact on this group's software development process.

In this case study, we treat a garbage collector as a component of a reactive system that also includes a heap object allocator and an abstraction of the user program. We use state transition system models and linear temporal logic for providing a specification and verification framework. Similar approaches have been successfully used in previous work on mechanically verifying garbage collectors (see Sec. 10). We introduce two systems at different levels of abstraction that we show to be related by a refinement relation. The more abstract system is simpler to verify, and the more concrete system is a more faithful model of an actual memory management system.

We chose to look at an algorithm that is relatively straightforward to analyse, so that we could quickly gain experience with the styles of proofs needed for reasoning about garbage collectors in linear temporal logic. In future work we will be studying the verification of successively more complicated algorithms.

Much of the literature on verifying garbage collection algorithms has focussed on abstract concurrent algorithms that have particularly subtle behaviours because of the fine-grain of the concurrency. Our interest at the moment is primarily in sequential algorithms, since few algorithms in use today are truly concurrent. At an abstract level, sequential garbage collection algorithms have simpler behaviours, but there are still plenty of challenges to be faced in verifying them, especially when considering implementation details. The techniques we are using for reactive systems are designed for reasoning with concurrency, and it would be easy to adapt our work to both concurrent and distributed settings.

We carry out our formalization using the PVS theorem prover [ORS92]. The PVS specification language is a classical higher-order logic with subtyping by arbitrary predicates. Proofs are carried out interactively by users repeatedly applying *strategies*, PVS's version of tactics. PVS has strategies for such operations as case splitting, expanding definitions, instantiating quantifiers, rewriting and simplifying. It also has strategies which invoke decision procedures that integrate congruence closure with linear arithmetic.

To accurately model garbage collection algorithms, we want to consider arbitrarily large and complex heap data structures. There is no obvious way to abstract these structures to produce finite state models suitable for model checking. Work so far in model checking garbage collection algorithms has had to use small fixed values (4 heap objects, for example) for heap parameters [Bru97, Hav96]. However, PVS has interfaces to several model checkers, and we hope in future to look at using model checking to assist in parts of our interactive proofs.

The rest of this paper is organized as follows: we present our formalization of linear temporal logic in Sec. 2 and then Sections 3, 4, and 5 describe the more abstract model we set up and show the safety and liveness proofs we carried out. Sec. 6 introduces our more concrete model, Sec. 7 summarises our framework for reasoning about refinement, and Sec. 8 covers the proof of refinement and the transfer of properties from the more abstract model to the more concrete. Sec. 9 discusses issues raised by the case study that are not covered elsewhere and a comparison with related work is made in Sec. 10. Finally we give our conclusions in Sec. 11.

We present definitions and lemmas in syntax that is very close to the actual syntax of PVS. The main change is that we replace certain keywords, operators and identifiers with non-ASCII logical and mathematical symbols.

2 Linear Temporal Logic

In Sec. 2.1 we define a notion of transition system and introduce a shallow embedding of linear temporal logic operators into the PVS specification language. We then go on in Sec. 2.2 and Sec. 2.3 to describe the most important rules we used in our proofs. Our approach is most similar to that of Manna and Pnueli [MP91a, MP91b]. Much is standard, but, as far as we know, the emphasis in Sec. 2.3 on the use of leads-to-via constructs for reasoning about liveness is novel. See also Sec. 9.1 and Sec. 9.2 for a discussion of some more subtle issues we had to address in order to produce an embedding that was practically useful and well-suited to our particular needs.

2.1 Basics

We consider a transition system to be characterized by a type `State` of states, a type `TxLab` of labels for transition kinds, a collection of binary relations on states `tx`, indexed by labels in `TxLab`, a set of initial states `init`, and a subset `fair` of `TxLab` being those transitions on which a fairness requirement is imposed.

PVS doesn't permit us to form tuples or records including types, so instead we define the record type

```
TxSys : TYPE = [# tx : [TxLab→pred[[State,State]]],
                  init : pred[State],
                  fair : set[TxLab] #]
```

for the non-type components of transition systems that is implicitly parameterized by types `State` and `TxLab`. The notations `[`S`→`T`]` and`[`S`,`T`]` are for function and product types respectively. `pred[`T`]` and `set[`T`]` are both definitions for the type `[`T`→bool]`. Square brackets in PVS syntax are used both for the syntax of type constructors and for explicitly specifying instantiating expressions for parameters. Definitions and lemmas in PVS are grouped into modules called *theories* which can take parameters. Definitions of types and constants in PVS are parameterised by the parameters of the theory they are defined in. When those types and constants are used, the parameters can either be left implicit for the PVS type checker to infer or can be explicitly supplied, as with, for example, `set[TxLab]` above. All the constants defined below are implicitly parameterised by a type `State` of some transition system, and most are also implicitly parameterized by a type `TxLab` and an element of type `TxSys`.

State formulas are predicates on states. The type of state formulas is

```
SFmla : TYPE = pred[State]
```

Let `A,B,C,D,E` and `I` be state formulas. Temporal formulas are predicates on pairs of form `(`σ`,i)` where σ is a sequence of states and `i` a natural number indicating a distinguished position in σ.

```
TFmla : TYPE = pred[[sequence[State],nat]]
```

Here `sequence[`T`]` is a definition for `[nat→`T`]`. This characterization of temporal formulas permits the definition of past-looking temporal operators. Let `P` and `Q` be temporal formulas. The function `tfm` defined by

```
tfm(A)(σ,i) : bool = A(σ(i))
```

coerces a state formula to a temporal formula. We declare `tfm` to be a PVS *conversion*. The PVS type checker automatically inserts conversions as necessary, so we usually omit explicit mention of `tfm`.

The □ (for every future time) and ◇ (at some future time) temporal operators are defined as

```
□(P)(σ,i) : bool = ∀(j : {i...}) : P(σ,j)
◇(P)(σ,i) : bool = ∀(j : {i...}) : P(σ,j)
```

and an *until* operator is defined as

```
𝒰(P,Q)(σ,i) : bool =
   ∃(j : {i...}) : Q(σ,j) ∧ ∀ (k : {i..j-1}) : P(σ,k)
```

The integer subrange types `{i...}` and `{i..j-1}` are definitions from standard theories in PVS that are always loaded. We lift PVS's quantifiers and logical connectives pointwise to the `SFmla` and `TFmla` types, and overload identifiers. For example:

```
P ⊃ Q  ≡  λ(σ,i) : P(σ,i) ⊃ Q(σ,i)
```

The lifted quantifiers take lambda terms as arguments. For presentation purposes, we suppress the lambda symbol: we present `∀(λn : P)` as `∀n : P`, for example. We abbreviate `□(P ⊃ Q)` by `P ⇒ Q`.

Let `sys` be some element of the `TxSys` type. We define several binary relations based on `sys`.

```
step(s,t) : bool = ∃a : tx(sys)(a)(s,t)
possible_step : pred[[State,State]] = refl_cl(step)
fair_step(s,t) : bool = ∃a : fair(sys)(a) ∧ tx(sys)(a)(s,t)
```

Here `a` is of type `TxLab`, and `refl_cl` is a reflexive closure operator. PVS uses prefix function application notation rather than the common postfix 'dot' notation for record projection operators (`tx(sys)` rather than `sys.tx`, for example).

A *run* is an infinite sequence of states generated by following transitions of `sys` from some initial state.

```
run : TFmla = tfm(init(sys)) ∧ □taken(possible_step)
```

where

```
taken(tx)(σ,i) : bool = tx(σ(i),σ(i+1))
```

We allow for the system to take idling transitions; these simplify proving refinement relations between systems and also free us from needing to separately consider finite runs. A *computation* is a run in which fairness conditions are obeyed.

```
computation : TFmla = run ∧ fairseq
```

Here we have

```
fairseq : TFmla =
  □ ¬ □ (tfm(enabled(fair_step)) ∧ ¬ taken(fair_step))

enabled(tx)(s) : bool = ∃t : tx(s,t)
```

A computation is a run in which it is never the case that a fair step is always enabled but never taken. See Sec. 9.1 for a discussion of our choice of what constitutes a fair run.

A temporal formula is *temporally valid* if it holds for all sequences of states. Frequently, we are concerned only with whether a formula holds for all computations of a given transition system. In this case we say a temporal formula is *temporally program valid*. A state formula is *state program valid* if it holds in all states of all computations of some system.

```
tv(P) : bool = ∀σ : P(σ,0)
tpv(P) : bool = tv(computation ⊃ P)
spv(A) : bool = tpv(□ (tfm(A)))
```

2.2 Safety Reasoning

We say a state formula is *invariant* if it is true in all accessible states. Since the fairness conditions we consider constrain only infinite runs, not finite initial segments of runs, a state formula is invariant just when it is state program valid. We say a state formula is *inductive* if it is invariant and, furthermore, its validity can be established by induction on the transition relation of the system. The induction rule we establish is

```
ind_rule_1 : LEMMA
  (∀ (s: (init(sys))) : I(s))  ∧  leadsto(step)(I,I)
  ⊃ spv(I)
```

where

```
leadsto(T)(A, B): bool = ∀s,t : A(s) ∧ T(s, t) ⊃ B(t)
```

and `(init(sys))` is an abbreviation for the type `{s:State | init(sys)(s)}`. As with any shallow embedding of a logic, we express rules of linear temporal logic as lemmas.

Inductive invariants I are commonly conjunctions $I_1 \wedge \ldots \wedge I_n$ of invariant formulas I_j that are not themselves inductive. Conveniently, we can divide proofs of the induction rule premise `leadsto(step)(I,I)` into separate proofs of `leadsto(step)(I,`I_j`)` for each j. Initial conjectures of invariants being inductive are often false and one needs to go through several iterations of adding conjuncts until one finds an invariant that is indeed inductive. As in the safety proofs that Havelund carried out [Hav96], we carefully set up definitions so that new conjuncts can be added to a conjectured inductive invariant without having to modify any existing proofs of conjuncts being preserved by `step`.

2.3 Liveness Reasoning

Temporal formulas of form `A ⇒ (B 𝒰 C)` are ubiquitous in our liveness proofs. Such a formula can be read as "if the system is in a state satisfying `A`, it will remain in states satisfying `B` until eventually a state satisfying `C` is reached". More concisely, the formula can be read as "`A` leads to `C` via B", and we refer to such a formula as a *leads-to-via* formula. We usually omit the parentheses in leads-to-via formulas, since the operator $\mathcal{U}$ binds more tightly than $\Rightarrow$.

Leads-to-via properties for states related by a single transition are established using the rule

```
one_step_leadsto : LEMMA
  (∀s : A(s) ⊃ enabled(fair_step)(s))
  ∧ leadsto(step)(A, B)
  ⊃ tpv(A ⇒ A 𝒰 B)
```

We have numerous rules for chaining together the flows of control described by leads-to-via formulas. For example,

```
leadsto_tx_l_or : LEMMA
  tv( (A ⇒ B 𝒰 C)  ∧  (C ⇒ D 𝒰 E)
      ⊃ ((A ∨ C) ⇒ (B ∨ D) 𝒰 E) )
```

A particularly useful rule is the following induction rule used for proving termination of transition loops:

```
wf_leadsto_rule : LEMMA
  tv( (∀t : (A ∧ λs : ρ(s) = t)
             ⇒ B 𝒰 ((A ∧ λs : ρ(s) < t) ∨ C))
      ⊃ (A ⇒ B 𝒰 C) )
```

Here `<` is some well-founded order relation (always the usual `<` ordering on naturals in our case), and ρ is a rank function mapping states to the type the well-founded order relation is over.

We prefer working with leads-to-via formulas rather than the more common but less informative *leads-to* formulas of form `A ⇒ ◇B` which is equivalent to `A ⇒ True 𝒰 B`. The reason is that they allow us to factor reasoning about flow of control and about how certain properties remain unchanged in specified regions of the system. The relevant rule is:

```
leadsto_stable_augmentation: LEMMA
  stable?(D, B)
  ⊃ tpv( (A ⇒ B 𝒰 C)  ⊃  ((A ∧ D) ⇒ B 𝒰 (C ∧ D)) )
```

where

```
stable?(D,B) : bool = leadsto(step)(D ∧ B,D)
```

If we know that control state `A` always leads to control state `C` via control states specified by `B`, and a property of data `D` is stable in region `B` and established in control state `A`, then we can conclude that property `D` will still hold when we reach control state `C`. We haven't seen this factorization benefit of leads-to-via formulas pointed out in the literature (in [OL82, MP91b] or [CM88], for example).

3 More Abstract Transition System Model

We describe in this section a model of a stop-and-collect, non-copyng algorithm. We assume a single thread of control (no concurrency). We use the tricolour marking scheme first introduced for a concurrent algorithm [DLM$^+$78]. The extra complication of the tricolour marking scheme might seem unnecessary for our immediate needs. However the scheme is useful for incremental collectors working in a sequential single-process setting, and we intend to look at such collectors in the near future. The scheme can also be viewed as an abstraction of most copying algorithms. See [JL96] for further details on this marking scheme and on garbage collection algorithms in general.

The idea behind most garbage collection algorithms in use is to first mark all heap objects accessible from certain root objects. Then objects not marked are considered garbage and can be collected. With tricolour marking, the roots start off marked grey and the rest of the heap is white. Marking proceeds by greying the white children of grey objects and blackening objects which have no white children. (We consider an object A to be the child of an object B if some field of B contains a pointer to A.) Marking is complete when there are no grey objects left. In the setting we consider here, where there is no interleaving of marking and user program activity, it is easy to see that the garbage objects are just those that are left white.

The model is parameterized by a finite non-empty type `Node` of heap nodes. One node `rt` is distinguished as being the *root* node of the heap. We think of nodes as representing objects in the heap.

We model pointers between objects using a directed graph. We consider an edge from node `m` to `n` as indicating that there are one or more pointers between the objects `m` and `n`.

```
Heap : TYPE = pred[[Node,Node]]
```

We are assuming here that the heap memory is divided up into object-sized chunks rather than being a continuous sequence of addresses and are not considering such issues as fragmentation. Each object is either i) free (available for allocation) or ii) allocated and marked with a certain colour. There is no need to record colours for free objects.

```
Color : TYPE = {free, black, grey, white}
Marking : TYPE = [Node→Color]
```

Our linear temporal logic framework assumes that the control state of a system is simply one component of the complete state. We use four control states

```
Control : TYPE = {mutate, alloc1, alloc2, trace}
```

and the type of system states is defined as

```
State : TYPE = [# heap: Heap,
                  marking : Marking,
                  control : Control #]
```

Table 1: Transitions for More Abstract Model

Name	From	To	Description
mutator transitions:			
`add_edge`	`mutate`	`mutate`	Pick nodes m and n reachable from the root and add an edge from m to n if one isn't there already.
`remove_edge`	`mutate`	`mutate`	Pick nodes m and n reachable from the root and remove any edge from m to n.
allocator transitions:			
`alloc_call`	`mutate`	`alloc1`	Always enabled.
`alloc_ok`	`alloc1`	`alloc2`	Enabled if at least one free node.
`alloc_sat`	`alloc2`	`mutate`	Pick free node n, remove any out edges of n, add edge from root to n and make n black.
collector transitions:			
`gc_init`	`alloc1`	`trace`	Make root grey and every black node white.
`trace_node`	`trace`	`trace`	Pick grey node n, grey any white children of n and make n black.
`gc_end`	`trace`	`alloc2`	If no grey nodes, free every white node.

In what follows, variables `d` and `e` are of type `State`.

We define eight kinds of transitions divided into three categories. They are described informally in Tab. 1. The 'From' column indicates the control state in which each transition is enabled and the 'To' column the value that the control state is changed to if each transition is taken.

The mutator transitions provide an abstract model of the user program, the allocator transitions model the allocation procedure of the memory management system and the collector transitions the garbage collection procedure. We have the collector invoked from the allocation routine, since this is the most common practice. For generality's sake, we don't insist that collection wait for the heap to be exhausted. We place the fairness requirement on the last five of the transitions, since these are internal to the memory manager and are not considered to be initiated by the user program.

Formally, each transition is defined as a binary relation on states. For example, the formal definition for the `gc_init` step is

```
gc_init(d,e) : bool =
  at(alloc1)(d) AND
  e = d WITH [(control) := trace,
              (marking) := init_marking_for_gc(marking(d))]
```

where

```
init_marking_for_gc(s : Marking) : Marking =
  λn : IF n = rt THEN grey
       ELSIF black?(s(n)) THEN white
```

```
    ELSE s(n) ENDIF
```

The `WITH` construct is convenient syntax for the non-destructive update of fields of records and points in the domains of functions. In this case, the `control` and `marking` fields of the record `d` are being updated.

We consider the initial state of the system to be one in which control is in the `mutate` state, the root is black, the rest of the nodes are free and there are no edges between nodes.

4 Proof of Safety of More Abstract Model

The safety property we prove is that the garbage collection algorithm only collects unreachable objects in the heap. Formally, we assert that all white nodes are unreachable whenever the next transition might be the collector transition `gc_end` which frees all white nodes.

```
safety : SFmla =
  enabled(gc_end) ⊃ ∀n : white(n) ⊃ ¬ reachable(n)

safety_lemma : LEMMA spv(safety)
```

where

```
reachable?(d)(m) : bool = star(heap(d))(rt,m)
reachable(m)(d) : bool = reachable?(d)(m)
white(m)(d) : bool = white?(marking(d)(m))
```

and `star` is reflexive transitive closure.

We prove `safety_lemma` using the induction rule `ind_rule_1` introduced in Sec. 2.2. The invariant `safety` is not inductive, so we show it to be true as a consequence of the stronger invariant `inv` that is inductive. `inv` is the conjunction of the properties:

1. the root node is grey or black,
2. there is no edge from a black node to a white node
3. there is no edge from a non-free node to a free node
4. no node is coloured grey when control is not in the `trace` state.

The proofs showing that this conjunction is inductive draw on a couple of auxiliary lemmas:

```
reachable_not_free : LEMMA
  inv(d) ∧ reachable?(d)(n) ⊃ ¬ free?(marking(d)(n))

reachable_black_if_no_greys : LEMMA
  inv(d) ∧ reachable?(d)(n) ∧ ¬ (∃m : grey?(marking(d)(m)))
    ⊃ black?(marking(d)(n))
```

The proofs of these auxiliary lemmas involve applying general properties of `star`. In particular, the induction lemma

```
narrowing_of_change : LEMMA
  star(R)(x,y) ∧ P(x) ∧ ¬ P(y) ⊃ ∃u,v : R(u,v) ∧ P(u) ∧ ¬ P(v)
```

is helpful.

The proof of `inv` being inductive splits naturally into 32 cases, one for each choice of conjunct and transition. The `grind` strategy automatically solves 25 of these. `grind` combines all the strategies mentioned in Sec. 1 and is often used to completely prove simpler goals and subgoals. Of these 25 cases, 10 are very straightforward because the selected conjunct refers only to parts of the state unchanged by the selected transition. `grind` solves these by rewriting with the state change equation contained in the transition definition and simplifying the resulting expressions. The other 15 automatic cases are more interesting. `grind` generates case splits, some based on update expressions, and guesses instantiations of quantifiers. The PVS simplifier knows of the distinctness of elements of the `Color` and `Control` datatypes and uses congruence closure to simplify equalities involving expressions of type `Color` and `Control` to true or false. The remaining 7 cases involve an average of about 7 steps of manual proof guidance (excluding work involved in proving the auxiliary lemmas).

5 Proof of Liveness of More Abstract Model

The liveness condition we prove is that garbage nodes always eventually become free. The formal statement is

```
liveness : TFmla =
  allocs_keep_coming
  ⊃ ∀m : (at(mutate) ∧ garbage(m)) ⇒ ◇ free(m)

garbage_eventually_freed_a : LEMMA tpv(liveness)
```

where

```
allocs_keep_coming : TFmla =
  at(mutate) ⇒ at(mutate) 𝒰 at(alloc1)
free(m)(s) : bool = free?(marking(s)(m))
garbage(m) : SFmla = ¬ free(m) ∧ ¬ reachable(m)
```

and `m` is of type `Node`.

The `allocs_keep_coming` precondition expresses the need to assume that the user program allocates new storage with sufficient regularity. If the user program doesn't keep calling the allocator, then there is no guarantee that garbage collection will ever take place. The restriction that we only consider garbage when in the `mutate` state is a minor one. It simplifies the proofs. With a little more work we could relax it.

The main stages of the proof are as follows. The first two have to do with flow of control. They establish that if we start in the mutate state, we eventually call the collector, and then eventually complete tracing of the reachable objects in the heap.

```
mutate_to_trace : LEMMA
  tpv( allocs_keep_coming
       ⊃ (at(mutate) ⇒ at((:alloc1,alloc2,mutate:)) 𝒰 at(trace)) )

eventually_no_greys_in_trace : LEMMA
  tpv( (at(trace) AND exist_greys)
       ⇒ (at(trace) ∧ exist_greys)
       𝒰 (at(trace) ∧ ¬ exist_greys) )
```

where

```
exist_greys(d) : bool =  ∃m : grey?(marking(d)(m))
```

Each of these lemmas is proved using the `wf_leadsto_rule` introduced in Sec. 2.3. In the first lemma, the rank of the state is the number of free nodes, in the second, the number of non-black nodes.

We note that, if making a transition starting in any state except one in which `gc_end` is enabled, garbage always remains garbage.

```
garbage_stable : LEMMA
  stable?(garbage(m), at((:alloc1,alloc2,mutate:))
                      ∨ (at(trace) ∧ exist_greys) )
```

Using the `leadsto_stable_augmentation` lemma discussed in Sec. 2.3 and the above lemmas about flow of control and stability of garbage, we deduce that, if we start at a point in a trace where we are in a `mutate` state and node m is garbage, we will always eventually reach a state in which the `gc_end` transition is enabled and m is still garbage.

We prove an additional invariant

```
garbage_in_trace_is_white : LEMMA
  spv(garbage(m) ∧ at(trace) ⊃ white(m))
```

and a characterization of the effect of the `gc_end` transition

```
freeing_of_whites: LEMMA
  leadsto(step)(at(trace) ∧ ¬ exist_greys ∧ white(m)
                ,free(m))
```

from which we easily derive `tpv(liveness)`.

6 More Concrete Transition System Model

This model is close in spirit to some that we at Edinburgh discussed with the memory management group at Harlequin when learning about the systems that they develop. The main difference between it and the more abstract model is that here we consider the edges between heap nodes as being labelled with elements of a type `Label`. Multiple edges are allowed between two given nodes, providing they have distinct labels. We model the heap by a function of type

```
Heap : TYPE = [Node,Label→Lift[Node]]
```

where `[S,T→U]` is an abbreviation for the type `[[S,T]→U]` and `Lift[T]` is a PVS datatype with elements `bot` and `lift(t)`, t being an element of type T. The labels on edges from the root node `rt` can be thought of as the names of heap roots, for example, the names of CPU registers or static variables or the addresses of stack locations. The labels on edges from a non-root node can be thought of as the names or addresses of pointer-containing fields of the heap object represented by the node. If the value of the heap function on node m and label r is `lift(n)`, then we consider there to be a pointer labelled r from m to n; if the value is `bot`, we consider the pointer r from m to be null.

The `control` and `marking` components of the state have the same definitions as in the more abstract model. In particular, every node has one of the four same colours. We add a new component `label_arg` of type `Label` which we explain below.

Tab. 2 shows the transitions. The `read`, `write`, `drop`, and `del` transitions replace the `add_edge` and `remove_edge` transitions of the more abstract model, and the `gc_init`, `trace_node`, `gc_end` and `alloc_ok` transitions have virtually the same definitions. We think of the `read` transition as the reading into CPU register s of the pointer in the field u of the object pointed to by CPU register r. We think of the `write` transition as the writing of a pointer to the object n into field u of object m. The conditions under which `read`, `write`, `drop`, and `del` are enabled are perhaps more restrictive than might be desired. If we were to relax them, we would need to add a transition in the more abstract model that simultaneously adds and remove edges. This would involve a little extra work, but would be straightforward. We think of the `label_arg` component of the state as holding the address on the stack for the return value of an allocation function: the the `alloc_call` transition sets this address and the `alloc_sat` transition sets the value to a fresh object.

As an example, the transition relation for the `read` transition is formalized as `∃r,u,s : read(r,u,s)(d,e)`, where

```
read(r,u,s)(d,e) : bool =
  at(mutate)(d) ∧ val?(d(r)) ∧ val?(d(val(d(r)),u)) ∧ bot?(d(s))
  ∧ e = d WITH [(heap)(rt,s) := d(val(d(r)),u)]
```

Conversions are used here to abbreviate expressions: when conversions are inserted, `d(n,u)` becomes `heap(d)(n,u)` and `d(r)` becomes `heap(d)(rt,u)`.

7 Framework for Refinement

We consider a transition system B to be a refinement of a system A when we can exhibit an abstraction mapping ϕ (sometimes called a *refinement mapping* [Lam94]) mapping states of B to states of A, that when applied to any computation of B yields a computation of A. In a theory parameterized by systems A and B, we make the definition

```
refinement?(φ) : bool =
  ∀σb : b.computation(σb,0) ⊃ a.computation(map(φ,σb),0)
```

where `φ` has type `[StateB→StateA]` and `σb` has type `sequence[StateB]`.

Table 2: Transitions for More Concrete Model

Name	From	To	Description
mutator transitions:			
`read`	`mutate`	`mutate`	If there are heap pointers $\texttt{rt} \xrightarrow{r} m$, $m \xrightarrow{u} n$ and $\texttt{rt} \xrightarrow{s} \emptyset$ (pointer s from root is null), then update the heap so that $\texttt{rt} \xrightarrow{s} n$.
`write`	`mutate`	`mutate`	If there are heap pointers $\texttt{rt} \xrightarrow{r} m$, $\texttt{rt} \xrightarrow{s} n$ and $m \xrightarrow{s} \emptyset$, then update the heap so that $m \xrightarrow{s} n$.
`drop`	`mutate`	`mutate`	If there is a pointer $\texttt{rt} \xrightarrow{r} m$, then make it null.
`del`	`mutate`	`mutate`	If there are pointers $\texttt{rt} \xrightarrow{r} m$ and $m \xrightarrow{u} n$, then null the pointer u.
allocator transitions:			
`alloc_call`	`mutate`	`alloc1`	If $\texttt{rt} \xrightarrow{r} \emptyset$, then store the label r in a component of the state called `label_arg`.
`alloc_ok`	`alloc1`	`alloc2`	Enabled if at least one free node.
`alloc_sat`	`alloc2`	`mutate`	If there is some free node n, then null all pointers out of n, add a pointer $\texttt{rt} \xrightarrow{r} n$ where r is the value of `label_arg` and make n black.
collector transitions:			
`gc_init`	`alloc1`	`trace`	Make root grey and every black node white.
`trace_node`	`trace`	`trace`	Pick grey node n, grey any white children of n and make n black.
`gc_end`	`trace`	`alloc2`	If no grey nodes, free every white node.

Identifiers in PVS can take prefixes that specify which theory they are from and how the parameters to that theory are instantiated. Local abbreviations can be introduced for these prefixes. The prefixes `a.` and `b.` in this section specify parameters appropriate for systems A and B. Later on, we use prefix abbreviations to distinguish between identifiers with the same name but from different theories. To improve readability, we use prefixes more than is strictly necessary: PVS's type checker can often resolve ambiguities when prefixes are left out. For simplicity, we deviate slightly here from exact PVS syntax in that we use the same prefix for identifiers from closely-related theories.

Because we consider computations rather than runs in the definition of `refinement?`, system B can inherit all temporally program valid properties proven of system A.

```
tpv_refinement : LEMMA
  refinement?(φ) ∧ a.tpv(PA) ⊃ b.tpv(treify(φ,PA))
```

```
spv_refinement : LEMMA
  refinement?(φ) ∧ a.spv(AA) ⊃ b.spv(sreify(φ,AA))
```

where `PA` and `AA` are, respectively, temporal and state formulas of system A, and the reification functions have definitions:

```
sreify(φ,AA) : SFmla[StateB] = λsb :  AA(φ(sb))
treify(φ,PA) : TFmla[StateB] = λσb,i : PA(map(φ,σb),i)
```

We establish that an abstraction mapping ϕ characterizes a refinement in two stages:

1. we show that ϕ maps runs of B to runs of A by showing that ϕ is a *simulation*:

   ```
   simulation?(φ) : bool =
     (∀sb : b.init(tsb)(sb) ⊃ a.init(tsa)(φ(sb)))
     ∧ (∀sb,tb : b.accessible?(sb) ∧ b.step(sb,tb)
                  ⊃ a.possible_step(φ(sb),φ(tb)) )
   ```

2. we show that the abstraction of every computation of B satisfies the 'fair sequence' property of A (see the definition of `fairseq` in Sec. 2.1). A simple case when this is true is when

   (a) a fair step of A being enabled implies that a fair step of B is enabled, and

   (b) if a fair step is taken between two adjacent states in a run of system B, then a fair step can also be taken in system A between the abstractions of these states.

   These conditions are fulfilled by refinements such as the one we consider in Sec. 8, where the refinement involves a change of data representation, but no significant change of control structure.

Our definition of refinement is similar to that of Chou [Cho93]. In particular, Chou identifies the same simple case of when a simulation is a refinement.

8 Verification of More Concrete Model

Let us refer to the more abstract transition system model introduced in Sec. 3 as `sys1` and the more concrete in Sec. 6 as `sys2`. We show that `sys2` is a refinement of `sys1`, considering a `sys1` that has the same type `Node` and same node `rt` as `sys2`. We define an abstraction mapping `rmap` that forgets the `label_arg` component of the `sys2` state, and is the identity function on the `control` and `marking` components. A heap component `h2` of the `sys2` state is mapped to the `sys1` heap `edge?(h2)` where

```
edge?(h)(m,n) : bool = ∃u : h(m,u) = lift(n)
```

Showing that `rmap` defines a simulation relation between `sys2` and `sys1` is straightforward. At one point we need to exploit the fact that the simulation only quantifies over all accessible states of the concrete system, not all states. The relevant lemma is

```
alloc_sat_simulation_a : LEMMA
  sys2.accessible?(d2) ∧ sys2.alloc_sat(d2,e2)
  ⊃ sys1.alloc_sat(rmap(d2),rmap(e2))
```

We need the `accessible?` precondition to know that, when `alloc_sat` is enabled, the pointer from `rt` named by the value of `label_arg` is null. If it isn't, the `sys2` `alloc_sat` operation might also remove an edge from root in the heap graph, a behaviour that isn't simulated by the `sys1` `alloc_sat` operation. To establish that this pointer is null when `alloc_sat` is enabled, we prove by induction the stronger invariant that this pointer is null whenever control is not in the `mutate` state.

Establishing the relationships between the fair steps being taken and enabled is also straightforward. For the relationship between the fair steps being taken, we can reuse the specific facts about transitions simulating each other such as the one cited above.

We therefore have

```
rmap_is_refinement : LEMMA refinement?(rmap)
```

and consequently

```
sys2_safety_a : LEMMA sys2.spv(sreify(rmap, sys1.safety))
sys2_liveness_a : LEMMA sys2.tpv(treify(rmap, sys1.liveness))
```

These formulations of the safety and liveness results for `sys2` are not satisfactory because they refer to the `sys1` characterizations. We therefore apply lemmas such as

```
treify_until : LEMMA
  treify(φ, PA 𝒰 QA) = treify(φ,PA) 𝒰 treify(φ,QA)
```

to push the reification operators down the definitions of `sys1.safety` and `sys1.liveness`, and lemmas about reification of atomic predicates to arrive at

```
sys2_safety_b : LEMMA sys2.spv(sys2.safety)
sys2_liveness_b : LEMMA  sys2.tpv(sys2.liveness)
```

where the definitions of `safety` and `liveness` are similar to those for `sys1`, but only refer to components of the state and transitions of `sys2`.

9 Discussion

9.1 Fairness

We discuss here our choice of fairness condition introduced in Sec. 2.1.

Since our primary interest for the moment is in sequential rather than concurrent algorithms, we don't need fairness conditions to account for a scheduler being fair to different processes. Rather, the only need for fairness conditions is to rule out runs where the system idles indefinitely with control at some internal point of a memory management procedure and with some transition of that procedure enabled.

The fairness condition we use is often called weak fairness [Lam94] or justice [MP91b]. More precisely, using terminology from Manna and Pnueli's book [MP91b, pp132–134], the condition we use is *process justice*. Manna and Pnueli use for their own transition systems a different weak fairness condition they call *transition justice*. In our notation, this is

```
tj_fairseq : TFmla =
 ∀(a:TxLab): □ ¬ □ (enabled(tx(sys)(a)) ∧ ¬ taken(tx(sys)(a)))
```

Manna and Pnueli give an example of a run of a system that exhibits livelock and is transition just but not process just. They argue that a more detailed implementation of this system would also exhibit livelock and so, if process justice were to be adopted, this system would erroneously be shown to be livelock free.

However, there are also scenarios in which a run that exhibits livelock can be process just, but not transition just. For example, consider a system with two states S and T and two fair transitions $S \to S$ and $S \to T$ belonging to one process that are always enabled in state S. A run in which the system always stays in state S is process just, but not transition just for the transition $S \to T$. Manna and Pnueli in [MP91b] miss this point because they claim that process justice is a more restrictive than transition justice.

A key feature of this example is the single-step looping transition $S \to S$. Such transitions are not uncommon in the sequential models we consider in which multiple program steps are represented by single transitions (consider `trace_node` in Sec. 3). It also seems that scenarios like the one that Manna and Pnueli cite rely on their being more than one process. We therefore prefer to use process justice rather than transition justice.

9.2 Working with Temporal Logic Judgements

One problem with linear temporal logic is the failure of equivalence of implication at the temporal logic level and the metalogic level (PVS's boolean logic level): the assertion `tv(P ⊃ Q)` is strictly stronger than the assertion `tv(P) ⊃ tv(Q)`. We find that we get strong enough rules only if we link the premises and conclusions at the temporal level rather than the boolean level. For example, the lemma:

```
weak_leadsto_tx : LEMMA
  tv(A ⇒ B 𝒰 C)  ∧  tv(C ⇒ D 𝒰 E)
  ⊃ tv(A ⇒ (B ∨ D) 𝒰 E)
```

is true, but not useful. Instead, we need:

```
leadsto_tx : LEMMA
  tv( (A ⇒ B 𝒰 C)  ∧  (C ⇒ D 𝒰 E)
      ⊃ (A ⇒ (B ∨ D) 𝒰 E) )
```

It is awkward to directly use standard strategies to apply rules phrased in this way. The solution we adopt is to preprocess such lemmas by unfolding the semantic definitions for `tv`, ∧ and ⊃ in the top-level temporal structure. This exposes boolean-level structure that standard strategies can work with. We also apply similar preprocessing to goals such as `garbage_eventually_freed_a` in

Sec. 5. For convenience, we integrate this preprocessing into other strategies we have for applying lemmas and breaking down goals.

This solution might seem unaesthetic and ad-hoc, but it is similar in practice to an approach being explored for TLA [Lam94] by Merz [Mer]. There, the judgement `w` $\models$ `P` of a temporal formula `P` being true at *world* `w` is introduced. A world corresponds in our case to a pair `(`σ`,i)` of a state sequence σ and a position `i` and the judgement to the application `P(`σ`,i)`. The difference is that in Merz's approach the type of worlds is not concretely specified. More generally, proof systems for temporal and modal logics that use such judgements are increasingly attracting interest in both the computer science and the logic communities. See Gabbay's book on labelled deduction systems [Gab96], for example.

9.3 PVS Strategies

We found the automation provided by Pvs's strategies to be of significant help. In Sec. 4 we give a brief analysis of which kinds of automation are used where in a few of the safety proofs.

In the course of the work, we added several extra strategies to those that are supplied by default. Some are general purpose and involve simply sequencing existing strategies, providing alternative default arguments, or providing slightly different functionality. Others are specific to our formulation of transition systems and to this particular model. Most of these others are for expanding certain sets of definitions. A few combine definition expansion with application of particular lemmas, carrying out case splits, and simplifying.

We found the strategy collection for the current version of Pvs[2] to be weakest when it comes to quantifier instantiation. Quantifier instantiation is handled by the `inst?` strategy. `inst?` searches for instantiations by matching parts of quantified formula bodies against expressions found in the formulas of the current goal sequent. Unfortunately, it often guesses unhelpful instantiations. The `grind` strategy calls on `inst?`, and `inst?` is the most common cause of `grind` failing to completely prove a goal. Pvs users consequently often run `grind` twice, first with `inst?` disabled. This improves matters a bit, but there's still a significant problem. The Pvs developers at SRI are well aware of this problem and are experimenting with tracking the polarities of formulas involved in matches to increase the likelihood that `inst?` guesses useful instantiations.

In the course of our work on the case study described in this paper, we have been experimenting with our own variations on `inst?`. We have done this partly to improve `grind`'s behaviour, but also to improve `inst?`'s usefulness when used for single or multiple step chaining. For example, we modified `inst?` to seek instantiations from matching multiple parts of quantified formula bodies. This is important for applying transitivity lemmas, since Pvs doesn't have logic variables.

There is certainly much further to go in this direction. For example, the Isabelle and HOL communities have found model elimination tactics of great use. Such tactics effectively do multiple step of chaining in constrained ways.

[2] V2.1, released April 1997

10 Comparison with Related Work

The tricolour algorithm we use for this case study was first put forward as a concurrent algorithm [DLM+78]. Ben-Ari considered this algorithm to be one of most difficult concurrent algorithms ever studied and proposed a two colour algorithm with similar properties, but with what he considered to be a significantly simpler proofs of safety and liveness [BA84]. Later pencil-and-paper proofs pointed out flaws with Ben-Ari's proofs, but these too contained flaws, and there were no fully correct proofs until Russinoff did a mechanical formalization in NQTHM [Rus94]. More recently, Havelund redid the formalization of the safety proof in PVS [Hav96], and Havelund and Shankar [HS96] looked at how the safety proof could be better organised using refinement techniques.

The models in all the work cited above abstract away the notion of objects being free. Object allocation is not modelled as a separate activity from the mutator updating the heap and object collection is modelled as some operation that makes the object accessible from root. Effectively, free list management is lumped in with the mutator. This abstraction improves the tractability of the proofs but results in models where the collector has the pathological behaviour of marking 'free' objects during the tracing phase. There is a loss in clarity here of the connection between the models and any real implementation of them. In contrast, we have set up more concrete models that do have free objects and that are close in spirit to abstract descriptions of garbage collection systems used by software engineers at Harlequin.

In two other ways our models are more abstract than those of the two colour and tricolour concurrent algorithms. Firstly our models are significantly more non-deterministic: we leave open many details of the memory management algorithm that are unimportant for reasoning about its correctness, for example, the order in which the collectors considers nodes for greying and blacking. Secondly, since we don't have to model interleavings of atomic operations of different processes, we can create abstract models with single transitions that represent many atomic operations (consider for instance the `trace_node` transition described in Sec. 3).

Having to model atomic operations doesn't always place a ceiling on how abstract a transition system one can consider. For example, Havelund and Shankar in [HS96] started with an abstract initial system with just two transitions. However, their approach is tailored for safety reasoning; there is no way in their approach that liveness properties can be inherited down the chain of refinements of models.

Gonthier formally verified the safety of a much more detailed concurrent garbage collection algorithm that was used in an experimental concurrent version of Caml-light [Gon96]. The state-transition system model involved 63 transitions and the safety proof 46 invariants. Gonthier argued many subtleties in the algorithm only came to light because of the realistic detail included in the model. Gonthier used the TLP system, a Larch-based theorem prover for TLA. The TLP script for the proof was 22,000 lines long. Few would want to repeat Gonthier's achievement without much better proof automation.

11 Conclusions

We successfully verified safety and liveness properties of a tricolour garbage collection algorithm that is close in spirit to abstract descriptions of garbage collection systems used by software engineers at Harlequin. We found it necessary and useful to adapt the presentations of transition systems and linear temporal logic we found in the literature. Most notably, we chose a slightly different notion of fairness and we had to develop a calculus for liveness reasoning based on a refinement of the common 'leadsto' operator.

We found PVS to be a suitable and effective tool for carrying out this formalization. We appreciated the automation provided by its decision procedures and supplied strategies, and were able to develop both general purpose and domain specific strategies that significantly simplified proofs.

We plan in the future on tackling successively more complicated garbage collection algorithms. For example, we intend to look at incremental read-barrier collectors and generational collectors.

12 Acknowledgements

The models presented in Sec. 3 and Sec. 6 are closely related to models that were developed in collaboration with, most notably, Healf Goguen and Rod Burstall at Edinburgh and Richard Brooksby, formerly at Harlequin.

The author wishes to thank Shmuel Katz and N. Shankar for their advice on linear temporal logic and Stephen Bevan, Rod Burstall, Healf Goguen, Cliff Jones, Pekka Pirinen, Gavin Matthews, Brian Monahan and the anonymous referees for their helpful comments on earlier drafts of this paper.

References

[AH96] Rajeev Alur and Thomas A. Henzinger, editors. *Computer Aided Verification : 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*. Springer, July 1996.

[BA84] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[Bru97] Glenn Bruns. *Distributed Systems Analysis with CCS*. Prentice Hall Europe, 1997.

[Cho93] Ching-Tsun Chou. Predicates, temporal logic, and simulations. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications: 6th International Workshop, HUG '93.*, volume 780 of *Lecture Notes in Computer Science*, pages 310–323. Springer-Verlag, August 1993.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[Gab96] Dov M. Gabbay. *Labelled deductive systems*, volume 1 of *Oxford Logic Guides*. Oxford University Press (Imprint: Clarendon Press), 1996.

[Gon96] Georges Gonthier. Verifying the safety of a practical concurrent garbage collector. In Alur and Henzinger [AH96].

[Hav96] Klaus Havelund. Mechanical verification of a garbage collector. Available from `http://www.cs.auc.dk/~havelund/`, May 1996.

[HS96] Klaus Havelund and Natarajan Shankar. A mechanized refinement proof for a garbage collector. Available from `http://www.cs.auc.dk/~havelund/`, December 1996.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic memory Management*. John Wiley & Sons, 1996.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Mer] Stephan Merz. Yet another encoding of TLA in Isabelle. Available from `http://www4.informatik.tu-muenchen.de/~merz/isabelle/`. The encoding described by this note accompanies the Isabelle98 release.

[MP91a] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83:97–130, 1991.

[MP91b] Zohar Manna and Amir Pnueli. *Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1991.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992. See `http://www.csl.sri.com/pvs.html` for up-to-date information on PVS.

[Rus94] David M. Russinoff. A mechanically verified garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.