VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Machine-Assisted Reasoning about Concurrency and Dynamic Memory

Keith Bauer, Lindsay Groves, Ray Nickson
Victoria University of Wellington

Mark Moir
Sun Microsystems, Boston

Abstract

This is the final report for 2001 on the joint project between Victoria University and Sun Microsystems. The project is concerned with machine-assisted reasoning about non-blocking concurrent algorithms involving dynamic data structures. We present PVS formalisms for reasoning about concurrency based on IO Automata, and for reasoning about dynamic memory allocation based on work by Luckham and Suzuki.

We would like to thank Sun Microsystems Research Laboratories for their support in funding this project.

Contents

1 Introduction

Concurrent programs are notoriously difficult to reason about because of the need to consider all possible ways in which processes may interact, as well as considering the behaviour of individual processes. These problems are further compounded in concurrent programs that manipulate dynamic data structures, which are also difficult to reason about, even in the absence of concurrency, because of the possibility of aliasing, where more than one pointer refers to the same storage location.

The motivation for this project arose from the observation by Mark Moir, whose research involves non-blocking algorithms for concurrent manipulation of dynamic data structures, that many published algorithms in that field contained subtle errors, even when accompanied by rigorous proofs. He decided to explore the possibilities of doing fully-formal, machine-checked proofs for these algorithms, and enlisted the assistance of Lindsay Groves and Ray Nickson, whose work includes program verification and mechanical theorem proving. Keith Bauer was employed as a research assistant over the (Southern) summer of 2000-2001, and worked on the project for his BSc(Honours) research project during 2001.

An initial examination of the literature revealed very little literature dealing with verification of concurrent algorithms for manipulation of dynamic data structures, so we undertook to develop the required techniques by combining general techniques for verification of concurrent algorithms with techniques for verifying sequential algorithms for dynamic data structures. We took as a our goal the verification of the Snark [5] deque algorithm, as an algorithm of moderate complexity, involving the various ingredients of interest, and we chose to use PVS to mechanise our proofs. Our plan was to formalise and prove properties of a series of algorithms, starting from very simple concurrent algorithms and sequential algorithms for simple dynamic data structures, and moving on to algorithms with more complex combinations of these features, such as Snark. This approach was designed to allow us to experiment with the PVS specification language and prover on small problems, before moving on to more complex ones, to investigate techniques for reasoning about concurrency and dynamic data structures separately before combining them, and to ensure that we achieved something useful even if we did not complete the proof of the Snark algorithm (which we didn't).

This report summarises the main outcomes of our work during 2001. Section 2 outlines some of the foundations of this work. Section 3 presents a very simple example, showing how PVS can be used to model and reason about concurrency. Section 4 presents a more complex example, a k-exclusion algorithm, using the same approach. Section 5 introduces IO Automata and discusses how they can be encoded in PVS. Section 6 discusses the problems associated with reasoning about dynamic data structures, and describes a PVS theory based on the work of Luckham and Suzuki. This theory is illustrated in Section7 with a simple example of pushing elements onto a linked list. Section 8 gives a brief sketch of our theory for the Snark algorithm. We then present our conclusions and disucss future work in Section 9. The appendices contain further details of the PVS theories developed.

2 Foundations

This chapter briefly introduces the various foundations on which this project builds. We begin by discussing the problems of reasoning about concurrency, the state-based representation for concurrent programs, and approaches to reasoning about concurrency. We then say a little about PVS.

2.1 Concurrency

Concurrent algorithms can be divided into two major classes, according to the way that processes interact: event-based concurrency, where processes interact by sending and receiving messages; and state-based concurrency, where processes interact by accessing shared memory. Event-based concurrency has been widely studied, and various process algebras, such as Milner's CCS [16] and Hoare's CSP [9], have been developed to reason about such systems.

The kinds of algorithms we are interested in, involving concurrent manipulation of dynamic data structures, require state-based concurrency, which does not have such a well developed theory. It is possible to describe concurrent programs that manipulate dynamic data structures as event-based systems, by passing the data structures as arguments in messages, but this quickly becomes very complex and does not really reflect the way in which such algorithms are implemented or the way that we think about them.

A crucial decision in designing and reasoning about concurrent programs is what we consider to be an "atomic" action. An atomic action is an action that a process can perform without any possibility of interference or interruption from another process — we can thus consider the effect of a set of processes executing concurrently as being a sequence of atomic actions, each performed by one of the processes in the set. Just what we take as being an atomic action depends on the hardware support we assume to be available for synchronisation; the different assumptions being important because different algorithms can be implemented with different synchronisation primitives; for example, the Snark algorithm, discussed in Section 8, depends on the availability of DCAS operation (see [8] for a disucssion of the capabilities of various synchronisation primitives).

2.2 State-Based Program Representation and IOA

A common way to make the assumptions about atomicity explicit is to model concurrent processes as state machines or automata, in which each transition represents an atomic action. This is rather like describing programs as flowcharts, where the flow of control is described by explicit test and branch logic, rather than the more complex control structures found in high-level languages, with the added restriction that each test or operation must be atomic. A system of concurrent processes is then modelled as a set of automata where, at each step, one automaton in the set performs one transition. These processes can be combined to form a single non-deterministic automaton, in which each transition represents an atomic step taken by one of the processes.

Several forms of state machines have been described in the literature, mostly varying only in minor details. In our first experiments, we used a formulation of state machines in PVS described by Shankar [20], and in later experiments we used Lynch and Tuttle's *IO Automata* [15]. IO Automata are a general formalism intended to allow a wide range of concurrent and distributed algorithms, and are supported by a specification language, IOA, and a suite of tools. We initially hoped to make use of some of these tools, but in the end we were not able to. We did, however, make use of the techniques described by Devillers [6] for translating IOA to PVS.

2.3 Reasoning about Concurrency

There are two standard ways to reason about the behaviour of concurrent programs. In the *invariant method*, we show that certain properties hold for all executions of the program. This can be done either by attaching assertions to specific points in the program, or to states in the automaton modelling the program, or by stating invariants that must hold at all points in the program, or all states in the automaton (see [12]). Since many of the required conditions need to hold at more than one place/state in the program, the latter is usually preferred, and the resulting invariants include so called *control predicates* for indicating the places in the program where various conditions are required to hold.

In the *simulation method*, we show that the behaviour of one automaton is consistent with that of a more abstract automaton, by demonstrating a correspondence between the states of the two machines. Sometimes the abstract automaton can be taken directly as a specification of the required behaviour, or we may verify that it has the desire behaviour using the invariant method — or it may be a simulation of an even more abstract machine.

The invariant method tends to be best suited to fairly simple problems, and the simulation method for more complex ones. We therefore began by using the invariant method, and considered the simulation method later, in some cases doing proofs using both approaches.

2.4 PVS

PVS is a specification language and theorem prover intended to support program verification. To quote the PVS tutorial [4]:

> PVS is a *verification system*: an interactive environment for writing formal specifications and checking formal proofs. [...] PVS provides an expressive specification language that augments classical higher-order logic with a sophisticated type system containing predicate subtypes and dependent types, and with parameterised theories and a mechanism for defining abstract datatypes such as lists and trees.

The central concept of PVS is that of a *theory* — a collection of type and constant declarations, axioms and theorems, which can be parameterised by any PVS entity other than a theory. For each theorem in a theory, PVS maintains a “proof chain” — a hierarchy of dependencies on other theorems, including those from other theories. Before a theory is considered “complete”, all these dependencies must be fully resolved and the proofs completed.

Since PVS has such a powerful type system, type checking can not be done automatically by the system as it might be in a less expressive language. PVS therefore generates “type-checking conditions” for each expression whose type it is not able to determine automatically. It is the responsibility of the user to prove that these conditions do indeed hold, and they can form part of the dependency chain for a theorem or theory.

We chose to use PVS for a variety of reasons. It is widely used in industry (if perhaps more often for hardware than software), but is gaining popularity in the software verification community as well. It has been used, for example, to verify garbage-collection algorithms [11]. The higher-order logic and dependent types provide a very rich framework for specification.

The availability of existing research into the translation of IOA to PVS certainly influenced our decision; however had this been the sole factor we would probably have chosen the Larch prover [7], as the IOA toolset in development at MIT directly supports this prover. The Larch prover, however, is works in first-order logic and has a very poor user interface.

PVS's ability to use higher-order logic makes certain properties significantly easier to state — for example we can assert the existence of a function satisfying a certain property, or quantify over all possible computations an automaton can make.

The PVS user interface is relatively polished, supporting unlimited undo and LaTeX pretty-printing. PVS also sports a graphical proof-tree viewer, giving random access to sequents and proof commands, as well as a good overview of the structure of the proof and its progress.

PVS's parameterised theories allow us to avoid specifying the number of concurrent processes, so we prove the algorithms correct regardless of how many processes are in the system. This means that even issues which don't show up until many processes interact are caught. The Snark deque has at least one condition that can only occur when three processes are in the system; there may be even more complex situations nobody has considered.

PVS is an interactive prover; it doesn't attempt to provide a proof unaided. We felt that the kinds of reasoning we were asking for were likely to give a fully-automated theorem prover or model checker trouble — if nothing else, the sheer number of cases concurrency introduces would cause the amount of computing time required to become unreasonable, although many of these cases can be easily discharged with a little human insight. The interactivity provided also allows us quick insight as to what precisely is going on in a particular situation, and react accordingly.

The major drawback to using an interactive prover is the amount of human time required to do the proofs. PVS attempts to alleviate this problem by allowing user-defined *strategies* (also called tactics by other prover systems). Unfortunately, we had no success in defining PVS strategies general enough to remove much of the burden of doing these proofs. The only strategies we created that we were able to reuse were those performing the "rote" steps at the beginning of an invariant proof — expanding definitions; setting up induction and the like.

We won't give a detailed description of the PVS language here. With a few essential bits explained, the rest should be sufficiently self-explanatory for the casual reader, and the more serious reader will need to consult the PVS manual [18] anyway.

3 Time Goes Ever Onward

In this section, we present a simple example to demonstrate the basic approach to reasoning about concurrency using invariants, and to show how this can be formalised using PVS following the approach used in [20].

Consider a program consisting of ten copies of the following process:

```
    shared X := 0;
    x := 0;
    y := 0;
    while TRUE do
0:     X := X + 1;     /* atomic */
1:     x := X;
2:     y := X;
    od
```

The variable `X` is shared by all copies of the process, and the operation of incrementing `X` is considered to be atomic. The other operations do not need to be atomic, since each copy has its own copies of `x` and `y`, so no interference can occur, but that also means that we loose nothing by assuming that they are atomic.

If this process was executed on its own, we would expect the values of x and y to always be the same, except just before line 2, when x would be one greater than y, and the values of x (and of y) form a contiguous sequence of natural numbers. When we have several copies executing concurrently, any number of other processors can update X while a given process is at line 1 or 2, so value of X assigned to Y in line 2 may be greater than the value assigned to x in line 1, and the result of incrementing X in line 0 may be more than one greater than the previous result of executing this line.

Let us now see how we can formalise this reasoning. We model a single process by the automaton shown in Figure 1. The automaton for the program with all ten processes is obtained by taking a kind of ten-fold product of the automaton for one process, as follows:

For each process i (i.e. $i \in 0..9$ for this example), let Q_i be the set of states for process i, q_{I_i} be the initial state for process i and N_i be the set of transitions for process i. The automaton for the entire program has state space $Q_0 \times Q_1 \times ... \times Q_9$, initial state $\langle q_{I_0}, \cdots, q_{I_9} \rangle$, and for each transition (q_{i_x}, q_{i_y}), where $q_{i_x}, q_{i_y} \in Q_i$, the program automaton has a corresponding transition from each state $\langle q_{0_a}, \cdots, q_{i_x}, \cdots, q_{9_z} \rangle$ to the state $\langle q_{0_a}, \cdots, q_{i_y}, \cdots, q_{9_z} \rangle$. This automaton is non-deterministic, with different possible actions in each state corresponding to the different processes that could take a step from that state.

We now formalise this model as a PVS theory, called counter, as shown in Figure 2. This PVS theory declares types PC, to represent the possible values of the program counter, consisting of natural numbers less than or equal to 2, and state, to represent program states, consisting of four components: a vector of program counters, a single variable X, a vector of x values and a vector of Y values. The three vectors have one element for each of the 10 processes. There is only one X value, because this variable is shared.

It then imports a generic theory called state, instantiating it with the type state just defined. The theory state defines the predicates Run and Inv used in the counter theory to state invariant properties; see Appendix A.

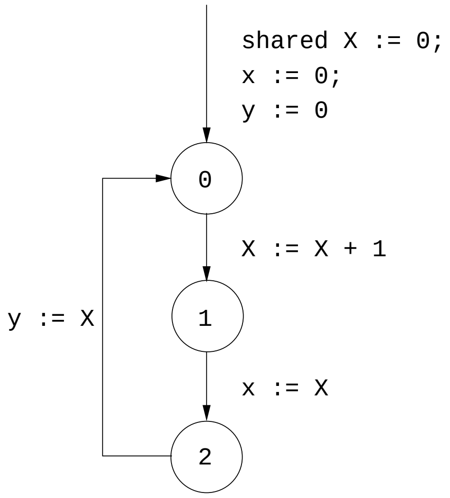

Figure 1: The Time Goes Ever Onward Automaton

The predicate InitState describes the initial state, and says that initially all variables are set to zero.

The relation Trans describes the possible transitions. It says that the system of processes can move from state s_0 to state s_1 if one of the processes can execute one of the three transitions shown in Figure 1; i.e. for some process, p, either:

- ps program counter in s_0 is 0, and in s_1, its program counter is 1 and the value of X is one greater than it was in s_0;
- ps program counter in s_0 is 1, and in s_1, its program counter is 2 and the value of x is the value that X had in s_0; or
- ps program counter in s_0 is 2, and in s_1, its program counter is 0 and the value of y is the value that X had in s_0.

Note how the WITH construct is used to describe updates to both state components and vectors; e WITH $[u_1 := v_1, \cdots, u_n := v_n]$ is the result of simultaneously replacing the components named or indexed by $u_1 \cdots u_n$ by the values of $v_1, \cdots, v_n$.

This way of describing transitions may seem cumbersome for a simple example like this, but proves to be more compact for larger examples, where there are more state components but only a few change on each transition.

With this theory, we can now prove the following properties of the concurrent program:

- Immediately before executing line two, $\mathtt{x} \leq \mathtt{X}$.

  This is formalised in the conjecture auxinv, which says that in any execution beginning in an initial state satisfying InitState and proceeding by performing transitions satisfying Trans, the property seenx holds in all states, where seenx says that for every process, when the program counter (pc) is equal to 2, the value of x for that process is less than or equal to the value of X.

```
counter: THEORY
 BEGIN

  PC: TYPE = {x: nat | x ≤ 2}

  state: TYPE =
  [# pc: [below(10) → PC],
     X: nat,
     x: [below(10) → nat],
     y: [below(10) → nat] #]

  IMPORTING state[state]

  s, s0, s1: VAR state

  InitState(s): bool =
      X(s) = 0 ∧
       (∀ (p: below(10)): pc(s)(p) = 0 ∧ x(s)(p) = 0 ∧ y(s)(p) = 0)

  Trans(s0, s1): bool =
    (∃ (p: below(10)):
       (pc(s0)(p) = 0 ∧
        s1 = s0 WITH [X := X(s0) + 1, pc := pc(s0) WITH [(p) := 1]])
         ∨
       (pc(s0)(p) = 1 ∧
        s1 = s0 WITH [x := x(s0) WITH [(p) := X(s0)],
                      pc := pc(s0) WITH [(p) := 2]])
         ∨
       (pc(s0)(p) = 2 ∧
        s1 = s0 WITH [y := y(s0) WITH [(p) := X(s0)],
                      pc := pc(s0) WITH [(p) := 0]]))

  seenx(s): bool = (∀ (p: below(10)): pc(s)(p) = 2 ⊃ x(s)(p) ≤ X(s))

  auxinv: CONJECTURE
    (∀ (aa: computation): (Run(InitState, Trans)(aa) ⊃ Inv(seenx)(aa)))

  safe(s): bool =
      (∀ (p: below(10)):
         (pc(s)(p) = 0 ∨ pc(s)(p) = 1) ⊃ x(s)(p) ≤ y(s)(p))

  main: CONJECTURE
    (∀ (aa: computation): Run(InitState, Trans)(aa) ⊃ Inv(safe)(aa))
 END counter
```

Figure 2: Formalisation of *Time Goes Ever Onward* in PVS

- Except immediately before executing line two, $\mathtt{x} \leq \mathtt{y}$.

  This is formalised in the conjecture main, which says that in any execution beginning in an initial state satisfying InitState and proceeding by performing transitions satisfying Trans, the property safe holds in all states, where safe says that for every process, when the program counter (pc) is equal to 0 or 1, the value of x for that process is less than or equal to the value of y for that process.

These properties are not particularly significant, and their proofs are quite straightforward, but they do demonstrate that this formalisation allows us to reasoning about concurrent access to a shared variable.

4 k-Exclusion

We now look at an algorithm for k-exclusion, a generalisation of the familiar mutual exclusion problem, in which no more than k (of N) processes are permitted to be in their critical region at once. The algorithm is taken from [1], and is shown in Figure 3.

```
shared variable
   X : −1..k;                                        /* Counter of available slots */
   Q : 0..N − 1                                      /* Spin location */
initially
   X = k

process p                                            /* 0 ≤ p < N */
   while true do
0:    Remainder Section;
1:    Acquire(N, k + 1);                             /* Entry section of (N, k + 1)-exclusion */
2:    if fetch_and_increment(X, −1) = 0 then         /* No slots available */
3:       Q := p;                                     /* Initialize spin location */
4:       if X < 0 then                               /* Still no slots available - must wait *
5:          while Q = p do /* null */ od             /* Busy-wait until released */
      fi fi;
6:    Critical Section;
7:    fetch_and_increment(X, 1);                     /* Release a slot */
8:    Q := p;                                        /* Release waiting process (if any) */
9:    Release(N, k + 1)                              /* Exit section of (N, k + 1)-exclusion */
   od
```

Figure 3: k-exclusion algorithm

We consider a slightly simplified case, where $N = k + 1$ (i.e. all but one of the processes can be in their critical section at the same time), in which case the *Acquire* and *Release* steps become skip statements, which is why the automaton (shown in Figure 4) has several steps with no operation. This time the algorithm has a non-trivial control structure, and uses an atomic *fetch_and_increment* operation to achieve the required synchronisation. Conditions on edges are enclosed in square brackets.

The program, consisting of N copies of this automaton, combined in the way described in Section 3, can be modelled in much the same way as for the example in Section 3. The resulting PVS theory can be found in Appendix B.

The invariants proved, as given in [1], are:

I1: $|\{q :: q@\{2..8\}\}| \leq k + 1$

I2: $X = k - |\{p :: p@\{3..7\}\}|$

I3: $X < 0 \Rightarrow (\exists p :: p@3 \vee (p@\{4, 5\} \wedge Q = p))$

I4: $|\{p :: p@6\}| \leq k$

This example demonstrates the translation of a program with more a complex control structure and the use of conditions on transitions, and illustrates parameterisation on the number of processes. Although the size of the algorithm does have an effect on the amount of work required for the proof, the increase is not unmanageable.

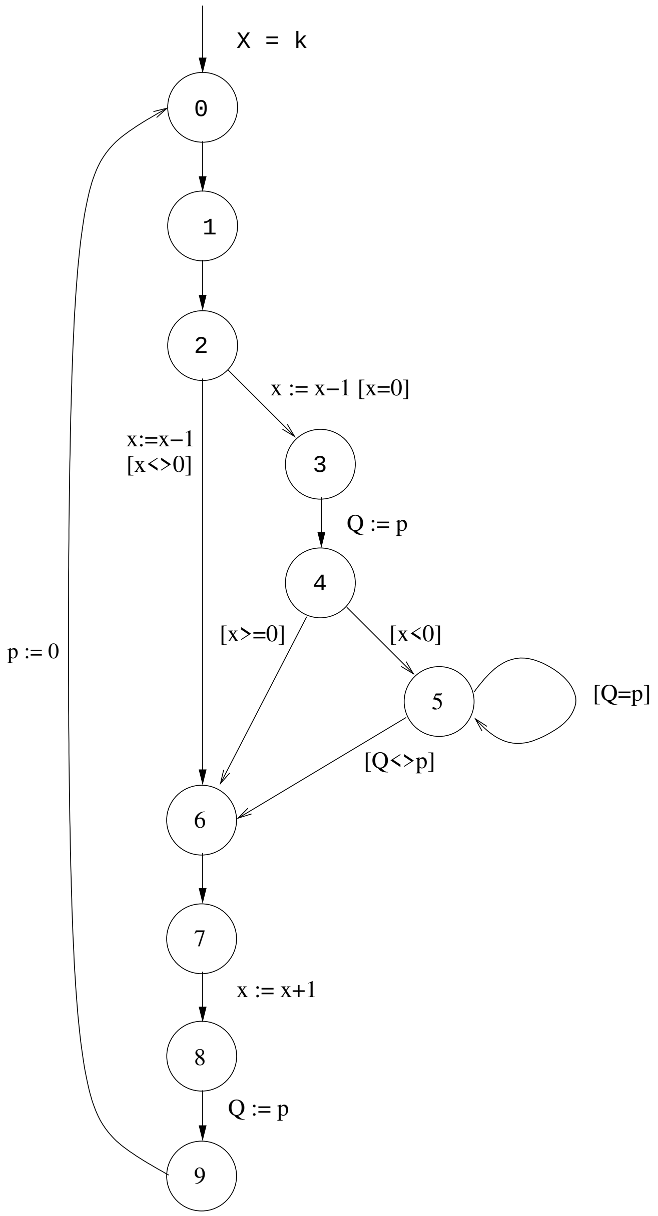

Figure 4: k-exclusion automaton

5 IO Automata in PVS

In this section, we explore the translation of IO Automata to PVS. We use an artificial automaton, as no example we have done illustrates all points clearly and simply.

An IO Automaton consists of five components [14, section 8.1]:

- a *signature*, $sig(A) = (in(S), out(S), int(S))$, defining the (disjoint) sets of input, output and internal actions of S,
- a set of *states*, $states(A)$,
- a nonempty set of *initial states*, $start(A) \subset states(A)$,
- a *state-transition relation*, $trans(A) \subset states(A) \times acts(A) \times states(A)$, where $acts(A) = in(S) \cup out(S) \cup int(S)$ and for every state s and input action π there is a transition (s, π, s') in $trans(A)$, and
- a *task-partition*, $tasks(A)$, which is a countable partition of $out(S) \cup int(S)$.

Appendix C shows a PVS theory defining the essential components of an IO Automaton, based on those given in [6]. This provides a type for defining the signature of an IO Automaton and functions for extracting the various components, and declares predicates reachable? and Invariant? for use in stating properties of IO Automata.

The theory is parameterised on the number of processes in the system, so that our reasoning is independent of the number of processes. For technical reasons we assume there is at least one process.[1]

The condition on the state-transition relation ensures that input actions are always enabled, which turns out to be irrelevant for put purposes as we make no use of input (or output) actions. Similarly, the task partition is irrelevant for our purposes.

We can now use this to define a theory for the automaton in question. We first declare a type with a value for each state a process can be in. Our hypothetical IO Automaton has three states, `A`, `B` and `C`.

```
Status: TYPE = {
  A, B, C
}
```

Next we declare a data type with a value for each transition a process can make. A PVS data type is an algebraic type specified by listing its constructors. It is assumed that the sequence of constructors used to create a value of this type is the unique way to represent that value. We must use a data type here, as we need to parameterise the actions by the process that takes them. Actions may also have additional parameters corresponding to inputs that action takes or outputs it makes. In this example, there are transitions from `A` to `B`, `B` to `C` and `C` to `AB`; the first transition takes an additional parameter, a natural number n. The forms with question marks after are *recognisers* for values of this data type constructed with the matching constructor.

[1]Specifically, we later have to prove the existence of a function having the Process type as its domain.

```
Action: DATATYPE BEGIN
  aAB(P: Process, n: nat): aAB?
  aBC(P: Process): aBC?
  aCA(P: Process): aCA?
END Action
```

Then we declare a tuple type holding the state for the automaton. There is a function *pc* (program counter) from processes to statuses, telling us the current state of each process. Each local variable is represented by a function from Process to values of the appropriate type. Each global variable is represented by a field of the appropriate type. In this example, we have a global variable X, a natural number, and a local variable y, holding an integer.

```
State: NONEMPTY_TYPE =
  [# pc: [Process → Status],
     X: nat,
     y: [Process → int] #]
```

We also declare the initial state the automaton is in by specifying a predicate that must be true of its initial state. We will assume that all processes starting state `A` with $y = 0$, and $X = 1$.

```
initial?(s): bool =
    ∀ (P: Process):
      (s`pc(P) = A) ∧ s`y(P) = 0) ∧
    s`X = 1
```

We then import the IOA theory. This is a little trickier than importing a normal theory due to a PVS issue to do with name conflicts.

```
automatonIOA: LIBRARY IOA[Action, State]
```

Every IO Automaton has a *signature*, specifying which actions are input actions (and therefore always enabled), which actions are output actions, and which actions are internal actions and only visible to this particular automaton. Since we are not dealing with communicating processes, all actions will be internal.

```
sig: signature =
    (# input? := (λ (a: Action): FALSE),
       output? := (λ (a): Action): FALSE),
       internal? := (λ (a): Action): TRUE) #)
```

We next define the semantics of each transition. Here we define the transition corresponding to the action aAB. It requires the process taking the transition to be in state A. After the transition, that process will be in state B, and its local variable y will have been assigned the value n.

```
tAB(s0, a, s1): bool =
      (∃ (P, n):
         a = aAB(P) ∧
         s0`pc(P) = A ∧
         s1 = s0 WITH [
           x := s0`y WITH [(P) := n],
           pc := s0`pc WITH [(P) := B]
         ])
```

The overall transition function for the automaton is defined by enumerating each of the individual transitions:

```
trans?(s0, a, s1): bool =
      action_p?(sig)(a) ∧
      (tAB(s0, a, s1) ∨
       tBC(s0, a, s1) ∨
       tCA(s0, a, s1))
```

Finally, we must create an IO Automaton from the signature, initial state and transitions described above.

```
io_automaton: IOAutomaton =
      (# sig := sig,
         lts := (# initial? := initial?, trans? := trans? #) #)
```

We can then state invariants of our automaton. The invariant is specified by a predicate on a state, and a corresponding lemma with a particular form. The Invariant? predicate quantifies the given predicate over all reachable states of the given automaton.

```
Trivial(s): bool =
  s`X > 0

Inv_Trivial: LEMMA
  Invariant?(io_automaton)(Trivial)
```

An IO Automata version of the k-exclusion algorithm is given in Appendix D. The linked list example (see Section 7 and Appendix F) and the Snark deque theory (see Appendix G) are also based on the IO Automaton model.

6 Dynamic Memory

6.1 Reasoning About Dynamic Memory

Reasoning about dynamic memory has long been considered a difficult problem, and a wide variety of approaches have been proposed (see for example [3], [10], [13], [17], [19]).[2] The basic problems are due to aliasing. When a single object can have multiple distinct names, a single pointer assignment can totally change the topology of the heap, making it difficult to ascertain the precise effect of the assignment.

Aliasing also occurs when dealing with arrays, since, for example, `A[x]` may refer to the same piece of memory as `A[y]`. When dealing with arrays, however, each element has a canonical name — `A[i]` for some constant `i`. But when dealing with pointers, no such canonical name exists, and some elements may only be accessed indirectly or may not even be accessible at all.

Dealing with these problems is, in general, very difficult. For example, a garbage-collection algorithm must deal with an arbitrary (and possibly changing) topology. Our initial attempts at modelling dynamic memory involved recursively defined invariants describing the reachability of certain nodes in the heap, which is used in the verification of garbage-collection algorithms and others that must deal with a completely arbitrary and unknown heap structure. Nelson [17] provides a nice notation and some axioms, and we were considering using his system until we realised that the problem can be simplified.

The algorithms we want to verify all share a common property — the structure of the heap is well-defined, because it contains a representation of a well-known data structure. We therefore don't in fact need to deal with the generic techniques, but can use simpler techniques as long as we bear the burden of tracking the heap structure ourselves.

We therefore build the structure of the heap into an invariant of the program. It is relatively easy to prove that any individual step does not falsify this invariant, and we don't have to deal with any of the issues mentioned above.

Our approach is to prove that the code implementing each operation on the data structure is equivalent to a single atomic action, and that the linked representation used by the algorithm is equivalent to a sequential representation. To show that the linked representation is equivalent to a sequential representation, we keep *auxiliary variables* to store the sequential representation, and use an invariant of the algorithm to show that the two are always in correspondence.

To show that this invariant holds, we have to ensure that it is not invalidated by normal program operation, and also that the atomic update action preserves the invariant. At the point at which the atomic update is considered to occur, we also update the sequential auxiliary variable. This ensures the atomicity of the update, as the update of the abstract is guaranteed to be atomic.

This proof technique is well-suited to the Snark deque, as there is a well-defined atomic action in every Snark operation where the change becomes visible to the outside world. It would perhaps not be as well suited to proving a more traditional lock-based algorithm, as no atomic update would occur. The technique could be adapted to handle the situation,

[2] These were summarised in an earlier report [2].

however, by rewriting the representation invariant to hold only outside the critical section where the update occurs and proving the mutual exclusion of this critical section.

6.2 The *Luckham-Suzuki* Dynamic Memory Theory

Having freed ourselves of the burden of having to prove reachability invariants, we were able to settle on a very simple representation of dynamic memory. We built a dynamic memory theory for PVS based on the theory of Luckham and Suzuki [13] for the verification of Pascal programs. Although Luckham and Suzuki's formalism allows for arrays, records and pointers, we have restricted ourselves to records and pointers, as we had no need for arrays. The PVS theory could, however, easily be extended to handle arrays if required.

Luckham and Suzuki exploit the similarities between arrays, records and pointers to produce almost identical axiomatic semantics for each. In order to bring out these similarities, they consider the following relationships:

> An index is to an array as
> a field name is to a record as
> a pointer is to a *reference class*

A reference class is conceptually the set of all pointers to objects of a specific type, T. In Luckham and Suzuki's notation, this is written $P\#T$. This reference class will change with time, but at a given time only one such reference class will exist for any given T.

They then introduce notations for selection: $A[i]$ for arrays, $r.f$ for records, and $R{\subset}p{\supset}$ for reference classes; assignment: $\langle A, [i], x\rangle$ (equivalent to `A[i] := x;`), $\langle r, .f, x\rangle$ and $\langle R, {\subset}p{\supset}, x\rangle$; and a notion of extension — the addition of a new pointer to the domain of a reference class: $R \cup \{p\}$.

In addition, they provide a *pointerTo* predicate taking a pointer and a reference class, true *iff* the pointer is in the domain of the reference class.

These operations are governed by fourteen axioms, providing identities relating to operations being used in sequence — for example, they state that $\langle r, .f, x\rangle.f = x$, and that $\langle\langle r, .f, x\rangle, .g, y\rangle = \langle\langle r, .g, x\rangle, .f, y\rangle$ provided f and g are distinct field names.

In order to make the transition to PVS, we had to make a number of simplifying assumptions.

Firstly, the only data type we allow is a pointer to a record. It would appear to be impossible to coerce PVS's type system into providing us with a richer type system than this, at least without doing so much manually that the benefits of using PVS are lost.[3] This does not lose any expressiveness — it is still possible to do anything that would be possible if, for example, there were an integer data type — indeed, integers can be simulated by asserting as an axiom the existence of a mapping between integers and pointers such that all the mapped pointers are distinct.

[3] For example, consider an attempt to provide type-safe distinct record types (specified, say, by a set of pairs $\langle fieldname, pointertype\rangle$). We must then declare the pointer type before this record type, yet the pointer type must be parameterised on this record type, and we run into a circularity. PVS does not allow circular definitions for reasons of preserving soundness.

Secondly, as another consequence of this typing problem, we can only have a single kind of record. In order not to sacrifice expressiveness, we allow any record to have any number of fields with any name. This allows the simulation of different record types.

For example, Luckham and Suzuki's axiom 3 ($\langle r, .f, x\rangle.f = x$) is translated into PVS as:

record_ax_1: AXIOM record_select(record_assign(r, f, p), f) = p

Luckham and Suzuki state their axiom for `new` in terms of the extension of an existing reference class by a pointer represented by a fresh identifier. This property is impossible to state in PVS, so we use a system of two axioms:

p: VAR Pointer

R, S: VAR ReferenceClass

new(R): [Pointer, ReferenceClass]

new_ax_1: AXIOM new(R) = (p, S) $\supset$ S = reference_class_extend(R, p)

new_ax_2: AXIOM new(R) = (p, S) $\supset$ $\neg$ pointerTo(p, R)

The inclusion of these two axioms, and the omission of the array axioms gives us a theory of fourteen axioms, providing a simple and intuitive way to reason about dynamic memory (see Appendix E).

7 Linked Lists

This section describes a simple program involving dynamic memory, to illustrate the use of the theory described in Section 6.2.

We consider the following program which repeatedly pushes a new item onto the head of a linked list, and prove that this linked list matches a linear sequence, the head of the list matching the low indices of the sequence. This example only involves a single execution thread; we will consider how to combine dynamic memory and concurrency later.

```
    head := NULL;
    x := NULL;
    i := NULL;
    while(TRUE) do
0:      x := new;
1:      i := new;
2:      x->data := i;
3:      x->next := head;
4:      head := x;
    od
```

This translates to a state machine as in Figure 5.

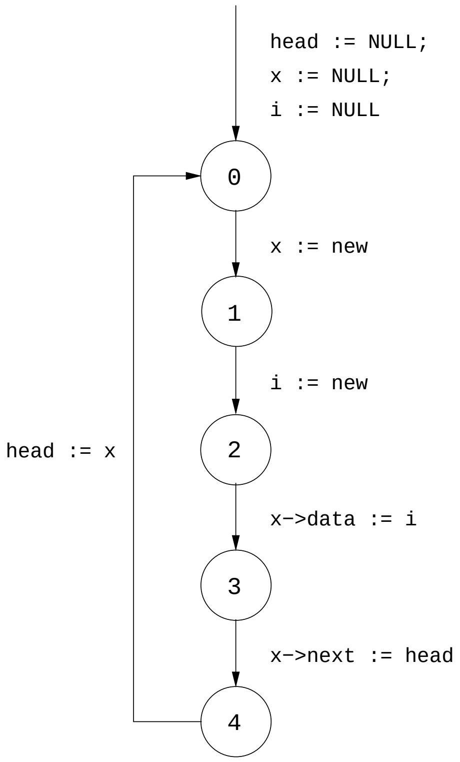

Figure 5: The Linked List Automaton

In order to facilitate the proof, we keep two auxiliary variables in the PVS theory (see Appendix F): `aux_seq` is the obvious sequential representation of the linked list, updated atomically at instruction four, and `aux_set` is the set of nodes that have been added to the list, also updated atomically at step four. This set is strictly unnecessary, but provides a convenient name for these nodes.

The main invariant to prove for this example is that the node representation of the linked list is equivalent to the sequential representation. To prove this, we assert that either:

- `head` is `NULL` and both the auxiliary set and the auxiliary sequence are empty; or
- there is a function f mapping indices of the auxiliary sequence to nodes in the auxiliary set such that:
  - for all indices i, $f(i)$`->data` is the value of the sequence at i,
  - if the index i is the last index in the sequence, $f(i)$`->next` is `NULL`, and
  - if the index i is not the last index in the sequence, $f(i)$`->next` is $f(i+1)$.

This ensures that the linked representation does indeed correspond to the sequential in the required manner — the first item in the sequence matches the item at the head of the linked list, and each subsequent sequence item matches the next item in the linked list.

This invariant is represented in PVS as follows:

```
s: VAR state

RepresentationOK(s): bool =
      (aux_seq(s) = empty_seq[Pointer] ∧ aux_set(s) = ∅[Pointer] ∧
       head(s) = NULL) ∨
       (∃ (f: [below[aux_seq(s)`length] → {p | (p ∈ aux_set(s))}]):
          ∀ (i: below[aux_seq(s)`length]):
            LET rfi: Record = reference_class_select(heap(s), f(i)) IN
              (record_select(rfi, "data") = aux_seq(s)`seq(i)) ∧
               IF i = aux_seq(s)`length − 1
                  THEN record_select(rfi, "next") = NULL
               ELSE record_select(rfi, "next") = f(i + 1)
               ENDIF
                ∧ (pointerTo(f(i), heap(s))) ∧ (head(s) = f(0)))

Inv_RepresentationOK: LEMMA Invariant?(ll_aut)(RepresentationOK)
```

In order to prove this invariant, a number of further invariants must be proved. All simply assert that variables don't change their values, except for the `MemoryOK` invariant, which ensures that all pointers are either `NULL` or have been allocated in the heap.

```
MemoryOK(s): bool =
      pointerTo(x(s), heap(s)) ∧
       pointerTo(i(s), index_heap(s)) ∧
```

```
    pointerTo(head(s), heap(s)) ∧
     (∀ p: (p ∈ aux_set(s)) ⊃ pointerTo(p, heap(s))) ∧
      (∀ (i: below[aux_seq(s)`length]):
         pointerTo(aux_seq(s)`seq(i), index_heap(s)))

Inv_MemoryOK: LEMMA Invariant?(ll_aut)(MemoryOK)
```

8 Snark

The Snark [5] deque algorithm uses the DCAS (double compare-and-swap) operation in conjunction with a reliance on garbage-collected dynamic memory allocation to provide a non-blocking implementation of a concurrent deque. It is a moderately complex algorithm, and thus provides a good example of an algorithm involving concurrent access to a shared dynamic data structure. It also illustrates the subtlety of such algorithms, and the need for fully formal verification: although the authors of the Snark algorithm had attempted a proof and in their article present an "outline of a proof sketch", but the algorithm as presented is in fact incorrect.

The essence of the Snark algorithm is a clever linked representation for deques, which allows uniform treatment of various special cases, making it easier to allow the visible effect of a push or pop operation to be performed by a single atomic action. We will not describe the full Snark algorithm here; we will just describe how one of the four operations, pushRight, is encoded in PVS, using the IOA theory described in Section 5.

The `pushRight` operation is define in [5] as follows:

```
 1  val pushRight(val v) {
 2    nd = new Node();  /* Allocate new Node structure */
 3    if (nd == Null) return ‘‘full’’;
 4    nd->R = Dummy;
 5    nd->V = v;
 6    while (true) {
 7      rh = RightHat;
 8      rhR = rh->R;
 9      if (rhR == rh) {
10        nd->L = Dummy;
11        lh = LeftHat;
12        if (DCAS(&RightHat, &LeftHat, rh, lh, nd, nd))
13          return ‘‘ok’’;
14      } else {
15        nd->L = rh;
16        if (DCAS(&RightHat, &LeftHat, rh, lh, nd, nd))
17          return ‘‘ok’’;
18  } } }
```

This is translated into the automaton shown in Figure 6. The condition `rhat == rh and lhat == lh` and action `rhat = nd; lhat = nd` on the transition from `sPushR10_11` to `sIdle` corresponds to the DCAS in line 12, and the action `aux_seq = aux_seq ++ <nd->v>` updates the auxiliary variable `aux_seq` by appending `v`. Similarly with the label on the transition from `sPushR15` to `sIdle`.

The automaton for leftPush is symmetrical to the one for rightPush, and the automata for leftPop and rightPop are similar. These automata are then combined into a single automaton, linked by the commom state common `sIdle`, so as to simulate the behaviour of an arbitrary program using the deque by nondeterministically performing the four operations in any order.

The resulting automaton can then be coded in PVS using the techniques described in

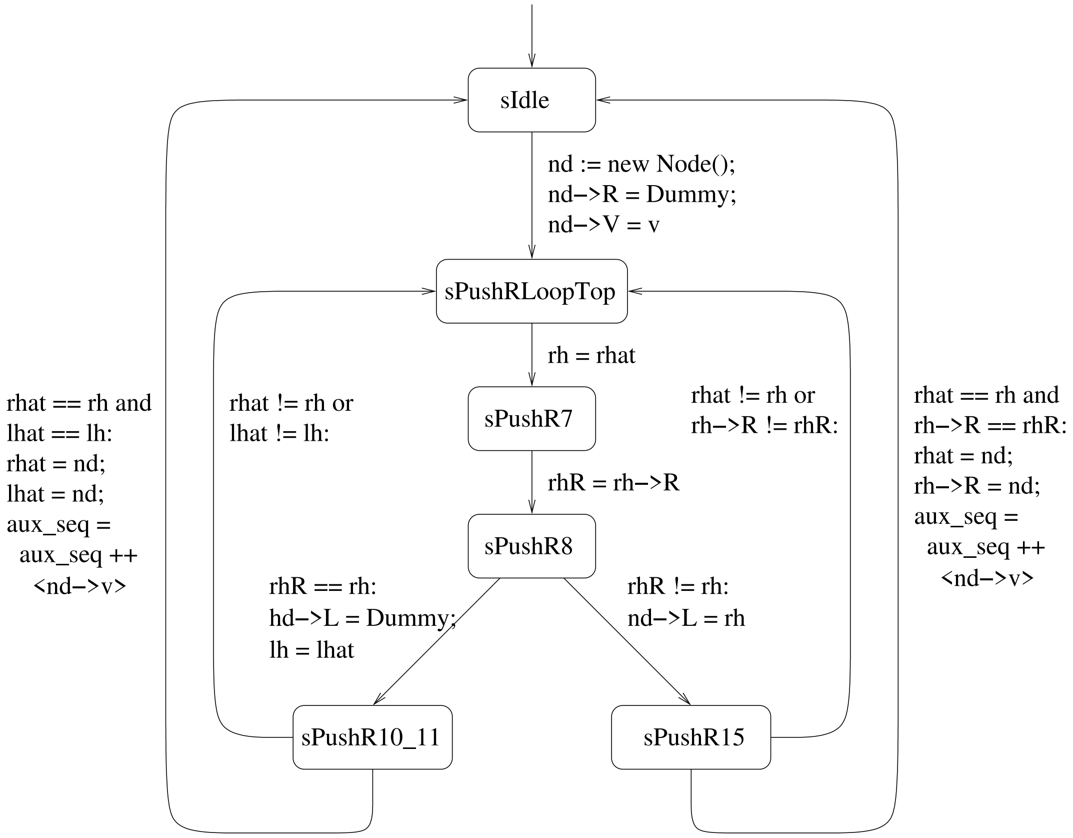

Figure 6: The Snark PushRight Automaton

Sections 5 and 6.2, as shown in Appendix G. This is considerably larger than our previous examples, but the elements are now almost all familiar. In addition to the IOA and dynamic memory theories, we use a new theory to describe the effects of the deque operations on an abstract sequence representation of the deque (Section G.2). The required invariant is to show that the linked representation and this absgract representation correspond, in the same was as for the linked list example in Section 7.

Although we have not completed the proof that this invariant holds, our experience with the previous examples suggests that the proof should be straightforward, if somehwat tedious.

There appears to be an error in Keith's Snark theory: The sequence of transitions for `popRight` *begins with* `tPopR3`*, which starts at state* `sPopRLoopTop`*, which can only be reached via transitions* `tPopREmpty` *and* `tPopR11`*, which are part of the* `popRight` *sequence. Similarly with* `popLeft`*.*

Is this something to do with pop not being applicable when the deque is empty?

Can we say something about the invariant, to illustrate the approach discussed earlier?

9 Conclusions and Future Work

We have been successful in achieving most of our goals for this work. We have found an easy and intuitive means of reasoning about concurrency in an algorithm, and have successfully integrated this approach into the PVS theorem prover.

We have successfully reasoned about programs involving dynamic memory, and shown that, for the algorithms of interest, it is not necessary to reason about reachability. It was therefore not necessary to deal with the more complex systems and formalisms proposed for this area, since a more straightforward and intuitive approach would suffice. This reduction of the problem from virtually intractable to nearly trivial gives us great hopes for the ease of proving other similar algorithms, and makes dynamic memory seem much less daunting.

Although, we did not complete the proof of the Snark deque algorithm, it served its rôle as a challenge problem admirably, by providing smaller targets to work on from which we learned a great deal about how to reason about concurrency and dynamic memory and how to be use PVS for this task. We are confident that what we have done will provide a good basis on which to complete that proof.

Translating programs into the PVS representation by hand was quite tedious, and a potential source of errors. It also limited the ease with which we could experiment with different algorithms or versions of algorithms. A tool such as the one described by Devillers [6] would be most helpful — although it was not a high priority for this project, it would be worth developing such a tool to assist in future work in this area, which will involve larger and more complex algorithms.

PVS proved to be adequate for our purposes, though not ideal. Most of the proofs require a large amount of user-direction, and although this was eased somewhat by the use of strategies, we found it hard to design strategies that could be reused on similar proofs.

The most creative part of any form of program verification is finding suitable invariants that guarantee the desired properties, and this is especially so in concurent algorithms where these invariants can be very subtle and sometimes quite unintuitive. PVS provides a rich specification language, with many different ways of expressing any given specification language, some of which may make it much easier or harder to express a given invariant, to prove the typing checking obligations, or to prove the invariants themselves. Learing how best to use such a language requires considerable experience, and we are clearly still learning.

As always, we may yet find better ways to approach various aspects of the problems we are considering. We intend to continue proving additional algorithms, while also looking at other approaches to reasoning about concurrency and dynamic memory to determine whether they would be useful and to better understand the pros and cons of the various approaches.

References

[1] James H. Anderson and Mark Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementation. *Distributed Computing*, 11(1):1–20, 1997.

[2] Keith Bauer, Lindsay Groves, Mark Moir, and Ray Nickson. Report on Sun/VUW Collaboration. Victoria University of Wellington, September 2001.

[3] Richard Bornat. Proving pointer programs in Hoare Logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[4] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. 1995.

[5] David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent deques. 2000.

[6] M. C. A. Devillers. Translating IOA automata to PVS. February 1999.

[7] Stephen Garland and John Guttag. A guide to LP: The larch prover. 1991.

[8] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):142–149, 1991.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[10] C. A. R. Hoare and He Jifeng. A trace model for pointers and objects. In R. Guerraoui, editor, *Proceedings of ECOOP '99*, number 1628 in Lecture Notes in Computer Science, pages 1–17. Springer Verlag, 1999.

[11] Paul B. Jackson. Verifying a garbage collection algorithm. 1998.

[12] Leslie Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, 1988.

[13] David C. Luckham and Norihisa Suzuki. Verification of array, record and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, October 1979.

[14] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[15] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[16] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[17] Greg Nelson. Verifying reachability invariants of linked structures. In *ACM Principles of Programming Languages*, pages 38–49, 1983.

[18] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Manual*. SRI International, Version 2.4, December 2001. (See `http://pvs.csl.sri.com/manuals.html`).

[19] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of Sir Tony Hoare*, pages 303–321, 2000.

[20] Natarajan Shankar. Machine-assisted verification using theorem proving and model checking. In Manfred Broy and Birgit Schneider, editors, *Mathematical Programming Methodology*, pages 499–528. Springer-Verlag, 1997. See `http://www.csl.sri.com/ shankar/shankar-papers.html`.

A PVS Theory of States

This theory is taken directly from [20], and is shown in the ASCII from used for input the PVS. The `state` theory is parameterised with a type, also called `state`, which is used to describe the state of a particular machine (see below), and imports the standard `sequences` theory, also parameterised with type `state`. Types `statepred`, `Action` and `computation` are then used to define predicates `Run` and `Inv`, which are used for stating properties

```
state[state: TYPE]: THEORY

 BEGIN

       IMPORTING sequences[state]

       statepred: TYPE = PRED[state]
       Action:    TYPE = PRED[[state,state]]
       computation: TYPE = sequence[state]

       pp:      VAR statepred
       action: VAR Action
       aa:      VAR computation

       Init(pp)(aa): bool = pp(aa(0))

       Inv(action)(aa): bool =
               (FORALL (n: nat): action(aa(n),aa(n+1)))

       Run(pp, action)(aa): bool =
               (Init(pp)(aa) AND Inv(action)(aa))

       Inv(pp)(aa) : bool =
               (FORALL (n: nat): pp(aa(n)))

 END state
```

B PVS Theory for k-Exclusion

```
kexcl[numprocs:nat, k:below(numprocs)]: THEORY

  BEGIN

    PC: TYPE = k: int | -1 <= k AND k <= 9
    PROC: TYPE = k: int | 0 <= k AND k <= numprocs-1
    state: TYPE = [# pc: [PROC->PC], X: int, Q: below(numprocs) #]

    IMPORTING card_extend[int, PROC]
    IMPORTING state[state]
    IMPORTING finite_sets@finite_sets_int
    IMPORTING finite_sets@finite_sets_card_eq[int, int]
    IMPORTING finite_sets@finite_sets_pred[PROC]

    s, s0, s1: VAR state
    p, q: VAR PROC

    at2_8(s,p): bool = member(pc(s)(p),x:PC|x >= 2 AND x <= 8)
    at3_7(s,p): bool = member(pc(s)(p),x:PC|x >= 3 AND x <= 7)
    at4_5(s,p): bool = member(pc(s)(p),x:PC|x >= 4 AND x <= 5)

    InitState(s): bool = X(s)=k AND (FORALL (p:PROC): pc(s)(p)=0)

    Trans(s0,s1): bool =
      (EXISTS (p:PROC):
          (pc(s0)(p)=0 AND
           s1 = s0 WITH [pc := pc(s0) WITH [(p) := 1]])
          OR

          (pc(s0)(p)=1 AND
           s1 = s0 WITH [pc := pc(s0) WITH [(p) := 2]])
          OR

          (pc(s0)(p)=2 AND
           s1 = s0 WITH [X := X(s0)-1,
                         pc := pc(s0) WITH [(p) := IF X(s0)=0 THEN 3 ELSE 6 ENDIF]])
          OR

         (pc(s0)(p)=3 AND
          s1 = s0 WITH [Q := p, pc := pc(s0) WITH [(p) := 4]])
         OR

         (pc(s0)(p)=4 AND
          s1 = s0 WITH [pc := pc(s0) WITH [(p) := IF X(s0) < 0 THEN 5 ELSE 6 ENDIF ]])
         OR
```

```
      (pc(s0)(p)=5 AND
       s1 = s0 WITH [pc := pc(s0) WITH [(p) := IF Q(s0)=p THEN 5 ELSE 6 ENDIF]])
      OR

      (pc(s0)(p)=6 AND
       s1 = s0 WITH [pc := pc(s0) WITH [(p) := 7]])
      OR

      (pc(s0)(p)=7 AND
       s1 = s0 WITH [X := X(s0)+1, pc := pc(s0) WITH [(p) := 8]])
      OR

      (pc(s0)(p)=8 AND
       s1 = s0 WITH [Q:=p, pc := pc(s0) WITH [(p) := 9]])
      OR

      (pc(s0)(p)=9 AND
       s1 = s0 WITH [pc := pc(s0) WITH [(p) := 0]])
   )

  I1(s): bool = card(q:PROC|at2_8(s,q)) <= k + 1

  Inv_I1: LEMMA  Invariant(InitState,Trans)(I1)

  I2(s): bool = X(s) = k - card(q:PROC|at3_7(s,q))

  Inv_I2: LEMMA Invariant(InitState,Trans)(I2)

  I3(s): bool = X(s) < 0 IMPLIES
           (EXISTS (p:PROC): pc(s)(p)=3 OR (at4_5(s,p) AND Q(s)=p))

  Inv_I3: LEMMA Invariant(InitState,Trans)(I3)

  I4(s): bool = card(p:PROC|extend[int,PROC,PC,-1](pc(s))(p)=6) <= k

  Inv_I4: LEMMA Invariant(InitState,Trans)(I4)

  X_OK(s): bool = -1 <= X(s) AND X(s) <= k

  Inv_X_OK: LEMMA Invariant(InitState, Trans)(X_OK)

END kexcl
```

C IOA PVS Theory

```
IOA[actions: TYPE+, states: TYPE+]: THEORY
 BEGIN

  trans_type: TYPE = [[states, actions, states] → bool]

  LTS: TYPE = [# initial?: pred[states], trans?: trans_type #]

  disjoint?(A, B, C: [actions → bool]): bool =
       (disjoint?(A, B) ∧ disjoint?(B, C) ∧ disjoint?(A, C))

  signature: TYPE =
  {s: [# input?: [actions → bool],
         output?: [actions → bool],
         internal?: [actions → bool] #] |
            disjoint?(input?(s), output?(s), internal?(s))}

  sig: VAR signature

  external_p?(sig): pred[actions] = (input?(sig) ∪ output?(sig))

  local_p?(sig): [actions → bool] = (internal?(sig) ∪ output?(sig))

  action_p?(sig): [actions → bool] =
       (external_p?(sig) ∪ internal?(sig))

  input_enabled_p?(lts: LTS, s: signature): bool =
       (∀ (s0: states, a: actions):
          input?(s)(a) ⇒ (∃ (s1: states): trans?(lts)(s0, a, s1)))

  IOAutomaton: TYPE =
  {M: [# sig: signature, lts: LTS #] | input_enabled_p?(lts(M), sig(M))}

  initial?: [IOAutomaton → pred[states]] =
       (λ (A: IOAutomaton): initial?(lts(A)))

  trans?: [IOAutomaton → trans_type] =
       (λ (A: IOAutomaton): trans?(lts(A)))

  input?: [IOAutomaton → pred[actions]] =
       (λ (A: IOAutomaton): input?(sig(A)))

  output?: [IOAutomaton → pred[actions]] =
       (λ (A: IOAutomaton): output?(sig(A)))

  internal?: [IOAutomaton → pred[actions]] =
```

$(\lambda\ (A\colon \text{IOAutomaton})\colon \text{internal?}(\text{sig}(A)))$

external?: [IOAutomaton $\to$ pred[actions]] =
$(\lambda\ (A\colon \text{IOAutomaton})\colon \text{external\_p?}(\text{sig}(A)))$

local?: [IOAutomaton $\to$ pred[actions]] =
$(\lambda\ (A\colon \text{IOAutomaton})\colon \text{local\_p?}(\text{sig}(A)))$

action?: [IOAutomaton $\to$ pred[actions]] =
$(\lambda\ (A\colon \text{IOAutomaton})\colon \text{action\_p?}(\text{sig}(A)))$

input_enabled?: [IOAutomaton $\to$ [LTS, signature $\to$ bool]] =
$(\lambda\ (A\colon \text{IOAutomaton})\colon \text{input\_enabled\_p?}(\text{lts}(A),\ \text{sig}(A)))$

CONVERSION trans?

CONVERSION initial?

CONVERSION input?

CONVERSION output?

CONVERSION internal?

CONVERSION external?

CONVERSION local?

CONVERSION action?

CONVERSION input_enabled?

A: VAR IOAutomaton

pp: VAR pred[states]

a: VAR actions

s, s_1: VAR states

reachable?$(A)(s)$: INDUCTIVE bool =
initial?$(A)(s) \lor (\exists\ (s_1,\ a)\colon$ reachable?$(A)(s_1) \land$ trans?$(A)(s_1,\ a,\ s))$

Invariant?(A)(pp): bool = $\forall\ (s)\colon$ reachable?$(A)(s) \supset$ (pp)(s)

END IOA

D IO Automaton for k-Exclusion

D.1 kexcl_automaton.pvs

```
kexcl_automaton[numprocs: {x: nat | x > 0}, k: below(numprocs)]: THEORY
 BEGIN

  IMPORTING common_types[numprocs, k]

  Status: TYPE = {idle, waiting, incs, leaving, left}

  states: TYPE+ = [# numincs: int, status: [PROC → Status], error: bool #]

  kexcl_IOA: LIBRARY IOA[actions, kexcl_automaton.states]

  a: VAR actions

  sig: signature =
       (# input? := (λ a: acquire?(a) ∨ release?(a)),
          output? := (λ a: grant?(a) ∨ ack?(a)),
          internal? := (λ a: do_release?(a)) #)

  s0, s1: VAR states

  acquire_t(s0, a, s1): bool =
       (∃ (p: PROC):
          a = acquire(p) ∧
           ((status(s0)(p) = idle ∧
             s1 = s0 WITH [status := status(s0) WITH [(p) := waiting]]) ∨
             (status(s0)(p) ≠ idle ∧ s1 = s0 WITH [error := TRUE])))

  grant_t(s0, a, s1): bool =
       (∃ (p: PROC):
          a = grant(p) ∧
           numincs(s0) < k ∧
            status(s0)(p) = waiting ∧
             s1 =
              s0
                WITH [numincs := numincs(s0) + 1,
                        status := status(s0) WITH [(p) := incs]])

  release_t(s0, a, s1): bool =
       (∃ (p: PROC):
          a = release(p) ∧
           ((status(s0)(p) = incs ∧
             s1 = s0 WITH [status := status(s0) WITH [(p) := leaving]]) ∨
             (status(s0)(p) ≠ incs ∧ s1 = s0 WITH [error := TRUE])))
```

```
  do_release_t(s0, a, s1): bool =
      (∃ (p: PROC):
         a = do_release(p) ∧
          status(s0)(p) = leaving ∧
           s1 =
            s0
              WITH [numincs := numincs(s0) − 1,
                    status := status(s0) WITH [(p) := left]])

  ack_t(s0, a, s1): bool =
      (∃ (p: PROC):
         a = ack(p) ∧
          status(s0)(p) = left ∧
           s1 = s0 WITH [status := status(s0) WITH [(p) := idle]])

  trans?(s0, a, s1): bool =
      action_p?(sig)(a) ∧
       ((error(s0) ∧ s0 = s1) ∨
         (¬ error(s0) ∧
           (acquire_t(s0, a, s1) ∨
             grant_t(s0, a, s1) ∨
              release_t(s0, a, s1) ∨
               do_release_t(s0, a, s1) ∨ ack_t(s0, a, s1))))

  initial?(s: states): bool =
      numincs(s) = 0 ∧ (∀ (p: PROC): status(s)(p) = idle) ∧ ¬ error(s)

  kex_aut: IOAutomaton =
      (# sig := sig, lts := (# initial? := initial?, trans? := trans? #) #)
 END kexcl_automaton
```

D.2 kexcl_props.pvs

```
kexcl_props[numprocs: {x: nat | x > 0}, k: below(numprocs)]: THEORY
 BEGIN

  IMPORTING finite_sets@finite_sets_int

  IMPORTING kexcl_automaton[numprocs, k]

  s, s0, s1: VAR states

  a: VAR actions

  p: VAR PROC
```

```
  incs(s): setof[PROC] = {p: PROC | status(s)(p) = incs}

  incs_leaving(s): setof[PROC] =
      {p: PROC | status(s)(p) = incs ∨ status(s)(p) = leaving}

  sublem: LEMMA card(incs(s)) ≤ card(incs_leaving(s))

  I_0(s): bool = error(s) ∨ numincs(s) = card(incs_leaving(s))

  I0_acq: LEMMA acquire_t(s_0, a, s_1) ∧ I_0(s_0) ⊃ I_0(s_1) ∨ error(s_1)

  I0_gra: LEMMA grant_t(s_0, a, s_1) ∧ I_0(s_0) ⊃ I_0(s_1) ∨ error(s_1)

  I0_rel: LEMMA release_t(s_0, a, s_1) ∧ I_0(s_0) ⊃ I_0(s_1) ∨ error(s_1)

  I0_ack: LEMMA ack_t(s_0, a, s_1) ∧ I_0(s_0) ⊃ I_0(s_1) ∨ error(s_1)

  Inv_I0: LEMMA Invariant?(kex_aut)(I_0)

  ltk(s): bool = numincs(s) ≤ k

  Inv_ltk: LEMMA Invariant?(kex_aut)(ltk)

  I_1(s): bool = error(s) ∨ card(incs(s)) ≤ k

  Inv_I1: LEMMA Invariant?(kex_aut)(I_1)
 END kexcl_props
```

D.3 common_types.pvs

```
common_types[numprocs: {x: nat | x > 0}, k: below(numprocs)]: THEORY
 BEGIN

  PROC: TYPE = below(numprocs)

  actions: DATATYPE
   BEGIN
    acquire(p: PROC): acquire?
    grant(p: PROC): grant?
    release(p: PROC): release?
    ack(p: PROC): ack?
    T_1(p: PROC): T1?
    T_2(p: PROC): T2?
    T_3(p: PROC): T3?
    T_4(p: PROC): T4?
    T_5(p: PROC): T5?
```

$T_7(p\colon \text{PROC})\colon$ T7?
$T_8(p\colon \text{PROC})\colon$ T8?
do_release(p: PROC): do_release?
END actions
END common_types

E Luckham-Suzuki PVS Theory

```
luckham_suzuki: THEORY
 BEGIN

  Pointer: NONEMPTY_TYPE

  NULL: Pointer

  p, q: VAR Pointer

  FieldName: TYPE = string

  f, g: VAR FieldName

  Record: NONEMPTY_TYPE

  r, s: VAR Record

  record_assign(r, f, p): Record

  record_select(r, f): Pointer

  record_ax_1: AXIOM record_select(record_assign(r, f, p), f) = p

  record_ax_2: AXIOM
    f ≠ g ⊃
      record_select(record_assign(r, f, p), g) = record_select(r, g)

  ReferenceClass: NONEMPTY_TYPE

  R, S: VAR ReferenceClass

  reference_class_assign(R, p, r): ReferenceClass

  reference_class_select(R, p): Record

  reference_class_extend(R, p): ReferenceClass

  reference_class_ax_1: AXIOM
    p = q ⊃
      reference_class_select(reference_class_assign(R, p, r), q) = r

  reference_class_ax_2: AXIOM
    p ≠ q ⊃
      reference_class_select(reference_class_assign(R, p, r), q) =
        reference_class_select(R, q)
```

```
reference_class_extend_ax_1: AXIOM
  reference_class_extend(reference_class_extend(R, p), q) =
   reference_class_extend(reference_class_extend(R, q), p)

reference_class_extend_ax_2: AXIOM
  p ≠ q ⊃
   reference_class_select(reference_class_extend(R, p), q) =
    reference_class_select(R, q)

reference_class_extend_ax_3: AXIOM
  p ≠ q ⊃
   reference_class_extend(reference_class_assign(R, p, r), q) =
    reference_class_assign(reference_class_extend(R, q), p, r)

pointerTo(p, R): bool

pointer_to_ax_1: AXIOM pointerTo(NULL, R)

pointer_to_ax_2: AXIOM pointerTo(p, reference_class_extend(R, p))

pointer_to_ax_3: AXIOM
  pointerTo(p, reference_class_assign(R, q, r)) = pointerTo(p, R)

pointer_to_ax_4: AXIOM
  p ≠ q ⊃
   (pointerTo(p, reference_class_extend(R, q)) = pointerTo(p, R))

pointer_to_ax_5: AXIOM ∀ R: ∃ p: ¬ pointerTo(p, R)

new(R): [Pointer, ReferenceClass]

new_ax_1: AXIOM new(R) = (p, S) ⊃ S = reference_class_extend(R, p)

new_ax_2: AXIOM new(R) = (p, S) ⊃ ¬ pointerTo(p, R)

END luckham_suzuki
```

F Linked List PVS Theory

```
linked_list: THEORY
 BEGIN

  IMPORTING luckham_suzuki

  p: VAR Pointer

  R: VAR ReferenceClass

  SOME_ARBITRARY_REFERENCE_CLASS: ReferenceClass

  PC: TYPE = {n: nat | 0 ≤ n ∧ n ≤ 4}

  action: DATATYPE
   BEGIN
    new_x: new_x?
    new_i: new_i?
    x_data_i: x_data_i?
    x_next_head: x_next_head?
    head_x: head_x?
   END action

  state: NONEMPTY_TYPE =
  [# pc: PC,
     heap: ReferenceClass,
     index_heap: ReferenceClass,
     head: Pointer,
     x: Pointer,
     i: Pointer,
     aux_seq: finite_sequence[Pointer],
     aux_set: finite_set[Pointer] #]

  IMPORTING finseq_additions[Pointer]

  llIOA: LIBRARY IOA[action, state]

  a: VAR action

  sig: signature =
      (# input? := (λ a: FALSE),
         output? := (λ a: FALSE),
         internal? := (λ a: TRUE) #)

  s, s0, s1: VAR state
```

```
Trans0(s0, a, s1): bool =
    new_x?(a) ∧
     pc(s0) = 0 ∧
      (∃ p, R:
         new(heap(s0)) = (p, R) ∧
          s1 = s0 WITH [pc := 1, heap := R, x := p])

Trans1(s0, a, s1): bool =
    new_i?(a) ∧
     pc(s0) = 1 ∧
      (∃ p, R:
         new(index_heap(s0)) = (p, R) ∧
          s1 = s0 WITH [pc := 2, i := p, index_heap := R])

Trans2(s0, a, s1): bool =
    x_data_i?(a) ∧
     pc(s0) = 2 ∧
      s1 = s0 WITH [pc := 3,
                    heap := reference_class_assign(heap(s0), x(s0),
                                 record_assign(reference_class_select(heap(s0), x(s0)),
                                 "data",
                                 i(s0)))]

Trans3(s0, a, s1): bool =
    x_next_head?(a) ∧
     pc(s0) = 3 ∧
      s1 = s0 WITH [pc := 4,
                    heap := reference_class_assign(heap(s0), x(s0),
                                 record_assign(reference_class_select(heap(s0), x(s0)),
                                 "next",
                                 head(s0)))]

Trans4(s0, a, s1): bool =
    head_x?(a) ∧
     pc(s0) = 4 ∧
      s1 = s0 WITH [pc := 0,
                    head := x(s0),
                    aux_seq := push_front(aux_seq(s0), i(s0)),
                    aux_set := (aux_set(s0) ∪ {x(s0)})]

trans?(s0, a, s1): bool =
    action_p?(sig)(a) ∧
     (Trans0(s0, a, s1) ∨
       Trans1(s0, a, s1) ∨
        Trans2(s0, a, s1) ∨ Trans3(s0, a, s1) ∨ Trans4(s0, a, s1))

initial?(s): bool =
    pc(s) = 0 ∧
```

```
  head(s) = NULL ∧
   x(s) = NULL ∧
    i(s) = NULL ∧
     aux_seq(s) = empty_seq[Pointer] ∧ aux_set(s) = ∅[Pointer]

ll_aut: IOAutomaton =
    (# sig := sig, lts := (# initial? := initial?, trans? := trans? #) #)

MemoryOK(s): bool =
    pointerTo(x(s), heap(s)) ∧
     pointerTo(i(s), index_heap(s)) ∧
      pointerTo(head(s), heap(s)) ∧
       (∀ p: (p ∈ aux_set(s)) ⊃ pointerTo(p, heap(s))) ∧
        (∀ (i: below[aux_seq(s)‘length]):
           pointerTo(aux_seq(s)‘seq(i), index_heap(s)))

Inv_MemoryOK: LEMMA Invariant?(ll_aut)(MemoryOK)

XNonNULL(s): bool = pc(s) ≠ 0 ⊃ x(s) ≠ NULL

Inv_XNonNULL: LEMMA Invariant?(ll_aut)(XNonNULL)

XUnreachable(s): bool =
    pointerTo(x(s), heap(s)) ∧
     (pc(s) = 1 ∨ pc(s) = 2 ∨ pc(s) = 3 ⊃ ¬ (x(s) ∈ aux_set(s)))

Inv_XUnreachable: LEMMA Invariant?(ll_aut)(XUnreachable)

XDataEqualsI(s): bool =
    (pc(s) = 3 ∨ pc(s) = 4) ⊃
     record_select(reference_class_select(heap(s), x(s)), "data") = i(s)

Inv_XDataEqualsI: LEMMA Invariant?(ll_aut)(XDataEqualsI)

XNextEqualsHead(s): bool =
    pc(s) = 4 ⊃
     record_select(reference_class_select(heap(s), x(s)), "next") =
      head(s)

Inv_XNextEqualsHead: LEMMA Invariant?(ll_aut)(XNextEqualsHead)

SeqEmptyHeadNULL(s): bool = aux_seq(s)‘length = 0 ⊃ head(s) = NULL

Inv_SeqEmptyHeadNULL: LEMMA Invariant?(ll_aut)(SeqEmptyHeadNULL)

RepresentationOK(s): bool =
    (aux_seq(s) = empty_seq[Pointer] ∧ aux_set(s) = ∅[Pointer] ∧ head(s) = NULL) ∨
     (∃ (f: [below[aux_seq(s)‘length] → {p | (p ∈ aux_set(s))}]):
```

```
∀ (i: below[aux_seq(s)`length]):
  LET rfi: Record = reference_class_select(heap(s), f(i)) IN
    (record_select(rfi, "data") = aux_seq(s)`seq(i)) ∧
     IF i = aux_seq(s)`length − 1
       THEN record_select(rfi, "next") = NULL
     ELSE record_select(rfi, "next") = f(i + 1)
     ENDIF
      ∧ (pointerTo(f(i), heap(s))) ∧ (head(s) = f(0)))

ROK0: LEMMA
  RepresentationOK(s0) ∧ Trans0(s0, a, s1) ⊃ RepresentationOK(s1)

ROK1: LEMMA
  RepresentationOK(s0) ∧ Trans1(s0, a, s1) ⊃ RepresentationOK(s1)

ROK2: LEMMA
  RepresentationOK(s0) ∧ reachable?(ll_aut)(s0) ∧ Trans2(s0, a, s1) ⊃
   RepresentationOK(s1)

ROK3: LEMMA
  RepresentationOK(s0) ∧ reachable?(ll_aut)(s0) ∧ Trans3(s0, a, s1) ⊃
   RepresentationOK(s1)

ROK4: LEMMA
  RepresentationOK(s0) ∧ reachable?(ll_aut)(s0) ∧ Trans4(s0, a, s1) ⊃
   RepresentationOK(s1)

Inv_RepresentationOK: LEMMA Invariant?(ll_aut)(RepresentationOK)
END linked_list
```

G Snark Deque PVS Theories

G.1 snark.pvs

```
snark[Process: NONEMPTY_TYPE]: THEORY
 BEGIN

  P: VAR Process

  IMPORTING luckham_suzuki

  p, q: VAR Pointer

  R, R0, R1, R2: VAR ReferenceClass

  SOME_ARBITRARY_REFERENCE_CLASS: ReferenceClass

  IMPORTING deseq[Pointer]

  d: VAR Deseq

  Action: DATATYPE
   BEGIN
    pushR1_5(P: Process, v: Pointer): pushR1_5?
    pushR7(P: Process): pushR7?
    pushR8(P: Process): pushR8?
    pushR10_11(P: Process): pushR10_11?
    pushR12(P: Process): pushR12?
    pushR15(P: Process): pushR15?
    pushR16(P: Process): pushR16?
    popR3(P: Process): popR3?
    popR4(P: Process): popR4?
    popR5(P: Process): popR5?
    popR6(P: Process): popR6?
    popREmpty(P: Process): popREmpty?
    popRNonEmpty(P: Process): popRNonEmpty?
    popR11(P: Process): popR11?
    popR12_15(P: Process): popR12_15?
    pushL1_5(P: Process, v: Pointer): pushL1_5?
    pushL7(P: Process): pushL7?
    pushL8(P: Process): pushL8?
    pushL10_11(P: Process): pushL10_11?
    pushL12(P: Process): pushL12?
    pushL15(P: Process): pushL15?
    pushL16(P: Process): pushL16?
    popL3(P: Process): popL3?
    popL4(P: Process): popL4?
```

```
  popL5(P: Process): popL5?
  popL6(P: Process): popL6?
  popLEmpty(P: Process): popLEmpty?
  popLNonEmpty(P: Process): popLNonEmpty?
  popL11(P: Process): popL11?
  popL12_15(P: Process): popL12_15?
 END Action

a: VAR Action

Status: TYPE =
 {sIdle, sPushRLoopTop, sPushR7, sPushR8, sPushR10_11, sPushR15,
  sPopRLoopTop, sPopR3, sPopR4, sPopR5, sPopREmpty, sPopRNonEmpty,
  sPopR10, sPopR11,
  sPushLLoopTop, sPushL7, sPushL8, sPushL10_11, sPushL15,
  sPopLLoopTop, sPopL3, sPopL4, sPopL5, sPopLEmpty, sPopLNonEmpty,
  sPopL10, sPopL11}

State: NONEMPTY_TYPE =
[# pc: [Process → Status],
   heap: ReferenceClass,
   DUMMY: Pointer,
   rhat: Pointer,
   lhat: Pointer,
   rh: [Process → Pointer],
   rhR: [Process → Pointer],
   rhL: [Process → Pointer],
   lh: [Process → Pointer],
   lhL: [Process → Pointer],
   lhR: [Process → Pointer],
   nd: [Process → Pointer],
   aux_seq: Deseq #]

s, s0, s1: VAR State

initial?(s): bool =
    ∀ P:
      (s`pc(P) = sIdle) ∧
       LET (p, R) = new(SOME_ARBITRARY_REFERENCE_CLASS) IN
         s`heap = R ∧ s`DUMMY = p

snarkIOA: LIBRARY IOA[Action, State]

sig: signature =
    (# input? := (λ a: FALSE),
       output? := (λ a: FALSE),
       internal? := (λ a: TRUE) #)
```

```
tPushR1_5(s_0, a, s_1): bool =
    (∃ P, q:
       a = pushR1_5(P, q) ∧
        s_0`pc(P) = sIdle ∧
         LET (p, R_0) = new(s_0`heap),
             R_1 = reference_class_assign(R_0, p,
                                  record_assign(reference_class_select(R_0, p), "R",
                                                s_0`DUMMY)),
             R_2 = reference_class_assign(R_1, p,
                                  record_assign(reference_class_select(R_1, p), "V", q))
           IN
           s_1 = s_0 WITH [heap := R_2,
                      nd := s_0`nd WITH [(P) := p],
                      pc := s_0`pc WITH [(P) := sPushRLoopTop]])

tPushR7(s_0, a, s_1): bool =
    (∃ P:
       a = pushR7(P) ∧
        (s_0`pc(P) = sPushRLoopTop) ∧
         s_1 = s_0 WITH [rh := s_0`rh WITH [(P) := s_0`rhat],
                    pc := s_0`pc WITH [(P) := sPushR7]])

tPushR8(s_0, a, s_1): bool =
    (∃ P:
       a = pushR8(P) ∧
        s_0`pc(P) = sPushR7 ∧
         LET p = record_select(reference_class_select(s_0`heap, s_0`rh(P)), "R") IN
           s_1 = s_0 WITH [rhR := s_0`rhR WITH [(P) := p],
                      pc := s_0`pc WITH [(P) := sPushR8]])

tPushR10_11(s_0, a, s_1): bool =
    (∃ P:
       a = pushR10_11(P) ∧
        s_0`pc(P) = sPushR8 ∧
         s_0`rhR(P) = s_0`rh(P) ∧
          LET R = reference_class_assign(s_0`heap, s_0`nd(P),
                                 record_assign(reference_class_select(s_0`heap,
                                                                      s_0`nd(P)),
                                               "L", s_0`DUMMY))
            IN
            s_1 = s_0 WITH [heap := R,
                       lh := s_0`lhat,
                       pc := s_0`pc WITH [(P) := sPushR10_11]])

tPushR12(s_0, a, s_1): bool =
    (∃ P:
       a = pushR12(P) ∧
        s_0`pc(P) = sPushR10_11 ∧
```

```
      IF (s0`rhat = s0`rh(P) ∧ s0`lhat = s0`lh(P))
        THEN s1 = s0 WITH [rhat := s0`nd(P),
                           lhat := s0`nd(P),
                           pc := s0`pc WITH [(P) := sIdle],
                           aux_seq := push_right(s0`aux_seq,
                               record_select(reference_class_select(s0`heap, s0`nd(P)),
                                             "V"))]
      ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPushRLoopTop]]
      ENDIF)

tPushR15(s0, a, s1): bool =
   (∃ P:
      a = pushR15(P) ∧
       s0`pc(P) = sPushR8 ∧
        s0`rhR(P) ≠ s0`rh(P) ∧
         LET R = reference_class_assign(s0`heap, s0`nd(P),
                      record_assign(reference_class_select(s0`heap,
                                                           s0`nd(P)),
                                    "L", s0`rh(P)))
           IN
           s1 = s0 WITH [heap := R, pc := s0`pc WITH [(P) := sPushR15]])

tPushR16(s0, a, s1): bool =
   (∃ P:
      a = pushR16(P) ∧
       s0`pc(P) = sPushR15 ∧
        IF (s0`rhat = s0`rh(P) ∧
              record_select(reference_class_select(s0`heap, s0`rh(P)), "R") = s0`rhR(P))
          THEN LET R = reference_class_assign(s0`heap, s0`rh(P),
                           record_assign(reference_class_select
                                            (s0`heap, s0`rh(P)),
                                         "R",
                                         s0`nd(P)))
                 IN
                 s1 = s0 WITH [heap := R,
                               rhat := s0`nd(P),
                               pc := s0`pc WITH [(P) := sIdle],
                               aux_seq := push_right(s0`aux_seq,
                                    record_select(reference_class_select(s0`heap,
                                                                         s0`nd(P)),
                                                  "V"))]
        ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPushRLoopTop]]
        ENDIF)

tPopR3(s0, a, s1): bool =
   (∃ P:
      a = popR3(P) ∧
       s0`pc(P) = sPopRLoopTop ∧
```

```
      s1 = s0 WITH [rh := s0`rh WITH [(P) := s0`rhat],
                    pc := s0`pc WITH [(P) := sPopR3]])

tPopR4(s0, a, s1): bool =
  (∃ P:
     a = popR4(P) ∧
      s0`pc(P) = sPopR3 ∧
      s1 = s0 WITH [lh := s0`lh WITH [(P) := s0`lhat],
                    pc := s0`pc WITH [(P) := sPopR4]])

tPopR5(s0, a, s1): bool =
  (∃ P:
     a = popR5(P) ∧
      s0`pc(P) = sPopR4 ∧
      IF (record_select(reference_class_select(s0`heap, s0`rh(P)), "R") = s0`rh(P))
        THEN s1 = s0 WITH [pc := s0`pc WITH [(P) := sIdle]]
      ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopR5]]
      ENDIF)

tPopR6(s0, a, s1): bool =
  (∃ P:
     a = popR6(P) ∧
      s0`pc(P) = sPopR5 ∧
      IF (s0`rhat = s0`rh(P) ∧ s0`lhat = s0`lh(P))
        THEN s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopREmpty]]
      ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopRNonEmpty]]
      ENDIF)

tPopREmpty(s0, a, s1): bool =
  (∃ P:
     a = popREmpty(P) ∧
      s0`pc(P) = sPopREmpty ∧
      IF (s0`rhat = s0`rh(P) ∧ s0`lhat = s0`lh(P))
        THEN LET (d, p) = pop_right(s0`aux_seq) IN
              s1 = s0 WITH [lhat := s0`DUMMY,
                            rhat := s0`DUMMY,
                            pc := s0`pc WITH [(P) := sIdle],
                        aux_seq := d]
      ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopRLoopTop]]
      ENDIF)

tPopRNonEmpty(s0, a, s1): bool =
  (∃ P:
     a = popRNonEmpty(P) ∧
      s0`pc(P) = sPopRNonEmpty ∧
      s1 = s0 WITH [rhL := record_select(
                            reference_class_select(s0`heap, s0`rh(P)), "L"),
                    pc := s0`pc WITH [(P) := sPopR10]])
```

```
tPopR11(s0, a, s1): bool =
    (∃ P:
        a = popR11(P) ∧
         s0`pc(P) = sPopR10 ∧
          IF (s0`rhat = s0`rh(P) ∧
                record_select(reference_class_select(s0`heap, s0`rh(P)), "L") = s0`rhL(P))
            THEN LET R = reference_class_assign(s0`heap, s0`rh(P),
                                        record_assign(reference_class_select
                                                            (s0`heap, s0`rh(P)),
                                                            "L",
                                                            s0`rh(P))),
                     (d, p) = pop_right(s0`aux_seq)
                   IN
                   s1 = s0 WITH [rhat := s0`rhL(P),
                                 heap := R,
                                 pc := s0`pc WITH [(P) := sPopR11],
                                 aux_seq := d]
          ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopRLoopTop]]
          ENDIF)

tPopR12_15(s0, a, s1): bool =
    (∃ P:
        a = popR12_15(P) ∧
         s0`pc(P) = sPopR11 ∧
          LET R = reference_class_assign(s0`heap, s0`rh(P),
                                  record_assign(
                                      reference_class_select(s0`heap, s0`rh(P)),
                                              "R", s0`DUMMY))
            IN s1 = s0 WITH [heap := R, pc := s0`pc WITH [(P) := sIdle]])

tPushL1_5(s0, a, s1): bool =
    (∃ P, q:
        a = pushL1_5(P, q) ∧
         s0`pc(P) = sIdle ∧
          LET (p, R0) = new(s0`heap),
              R1 = reference_class_assign(R0, p,
                           record_assign(reference_class_select(R0, p), "L",
                                               s0`DUMMY)),
              R2 = reference_class_assign(R1, p,
                           record_assign(reference_class_select(R1, p), "V", q))
            IN
            s1 = s0 WITH [heap := R2,
                          nd := s0`nd WITH [(P) := p],
                          pc := s0`pc WITH [(P) := sPushRLoopTop]])

tPushL7(s0, a, s1): bool =
    (∃ P:
```

```
      a = pushL7(P) ∧
       (s0‘pc(P) = sPushLLoopTop) ∧
        s1 = s0 WITH [lh := s0‘lh WITH [(P) := s0‘lhat],
                      pc := s0‘pc WITH [(P) := sPushL7]])

tPushL8(s0, a, s1): bool =
    (∃ P:
      a = pushL8(P) ∧
       s0‘pc(P) = sPushL7 ∧
        LET p = record_select(reference_class_select(s0‘heap, s0‘lh(P)), "L") IN
          s1 = s0 WITH [lhL := s0‘lhL WITH [(P) := p],
                        pc := s0‘pc WITH [(P) := sPushL8]])

tPushL10_11(s0, a, s1): bool =
    (∃ P:
      a = pushL10_11(P) ∧
       s0‘pc(P) = sPushL8 ∧
        s0‘lhL(P) = s0‘lh(P) ∧
         LET R = reference_class_assign(s0‘heap, s0‘nd(P),
                              record_assign(reference_class_select(s0‘heap,
                                                                   s0‘nd(P)),
                                            "R", s0‘DUMMY))
           IN
           s1 = s0 WITH [heap := R,
                         rh := s0‘rhat,
                         pc := s0‘pc WITH [(P) := sPushL10_11]])

tPushL12(s0, a, s1): bool =
    (∃ P:
      a = pushL12(P) ∧
       s0‘pc(P) = sPushL10_11 ∧
        IF (s0‘lhat = s0‘lh(P) ∧ s0‘rhat = s0‘rh(P))
          THEN s1 = s0 WITH [lhat := s0‘nd(P),
                             rhat := s0‘nd(P),
                             pc := s0‘pc WITH [(P) := sIdle],
                             aux_seq := push_left(s0‘aux_seq,
                                          record_select(
                                            reference_class_select(s0‘heap, s0‘nd(P)),
                                                  "V"))]
        ELSE s1 = s0 WITH [pc := s0‘pc WITH [(P) := sPushLLoopTop]]
        ENDIF)

tPushL15(s0, a, s1): bool =
    (∃ P:
      a = pushL15(P) ∧
       s0‘pc(P) = sPushL8 ∧
        s0‘lhL(P) ≠ s0‘lh(P) ∧
         LET R = reference_class_assign(s0‘heap, s0‘nd(P),
```

```
                                        record_assign(reference_class_select(s0‘heap,
                                                                             s0‘nd(P)),
                                                      "R", s0‘lh(P)))
         IN
         s1 = s0 WITH [heap := R, pc := s0‘pc WITH [(P) := sPushL15]])

tPushL16(s0, a, s1): bool =
   (∃ P:
      a = pushL16(P) ∧
       s0‘pc(P) = sPushL15 ∧
        IF (s0‘lhat = s0‘lh(P) ∧
             record_select(reference_class_select(s0‘heap, s0‘lh(P)), "L") = s0‘lhL(P))
          THEN LET R = reference_class_assign(s0‘heap, s0‘lh(P),
                                            record_assign(reference_class_select
                                                         (s0‘heap, s0‘lh(P)),
                                                         "L",
                                                         s0‘nd(P)))
               IN
               s1 = s0 WITH [heap := R,
                             lhat := s0‘nd(P),
                             pc := s0‘pc WITH [(P) := sIdle],
                             aux_seq := push_left(s0‘aux_seq,
                                                  record_select(
                                                     reference_class_select(s0‘heap,
                                                                            s0‘nd(P)),
                                                        "V"))]
        ELSE s1 = s0 WITH [pc := s0‘pc WITH [(P) := sPushLLoopTop]]
        ENDIF)

tPopL3(s0, a, s1): bool =
   (∃ P:
      a = popL3(P) ∧
       s0‘pc(P) = sPopLLoopTop ∧
        s1 = s0 WITH [lh := s0‘lh WITH [(P) := s0‘lhat],
                      pc := s0‘pc WITH [(P) := sPopL3]])

tPopL4(s0, a, s1): bool =
   (∃ P:
      a = popL4(P) ∧
       s0‘pc(P) = sPopL3 ∧
        s1 = s0 WITH [rh := s0‘rh WITH [(P) := s0‘rhat],
                      pc := s0‘pc WITH [(P) := sPopL4]])

tPopL5(s0, a, s1): bool =
   (∃ P:
      a = popL5(P) ∧
       s0‘pc(P) = sPopL4 ∧
        IF (record_select(reference_class_select(s0‘heap, s0‘lh(P)), "L") = s0‘lh(P))
```

```
      THEN s1 = s0 WITH [pc := s0`pc WITH [(P) := sIdle]]
    ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopL5]]
    ENDIF)

tPopL6(s0, a, s1): bool =
  (∃ P:
    a = popL6(P) ∧
     s0`pc(P) = sPopL5 ∧
      IF (s0`lhat = s0`lh(P) ∧ s0`rhat = s0`rh(P))
        THEN s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopLEmpty]]
      ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopLNonEmpty]]
      ENDIF)

tPopLEmpty(s0, a, s1): bool =
  (∃ P:
    a = popLEmpty(P) ∧
     s0`pc(P) = sPopLEmpty ∧
      IF (s0`lhat = s0`lh(P) ∧ s0`rhat = s0`rh(P))
        THEN LET (d, p) = pop_left(s0`aux_seq) IN
              s1 = s0 WITH [rhat := s0`DUMMY,
                            lhat := s0`DUMMY,
                            pc := s0`pc WITH [(P) := sIdle],
                            aux_seq := d]
      ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopLLoopTop]]
      ENDIF)

tPopLNonEmpty(s0, a, s1): bool =
  (∃ P:
    a = popLNonEmpty(P) ∧
     s0`pc(P) = sPopLNonEmpty ∧
      s1 = s0 WITH [lhR := record_select(reference_class_select(s0`heap, s0`lh(P)), "R"),
                    pc := s0`pc WITH [(P) := sPopL10]])

tPopL11(s0, a, s1): bool =
  (∃ P:
    a = popL11(P) ∧
     s0`pc(P) = sPopL10 ∧
      IF (s0`lhat = s0`lh(P) ∧
           record_select(reference_class_select(s0`heap, s0`lh(P)), "R") = s0`lhR(P))
        THEN LET R = reference_class_assign(s0`heap, s0`lh(P),
                                            record_assign(reference_class_select
                                                           (s0`heap, s0`lh(P)),
                                                           "R",
                                                           s0`lh(P))),
                 (d, p) = pop_left(s0`aux_seq)
              IN
              s1 = s0 WITH [lhat := s0`lhR(P),
                            heap := R,
```

```
                              pc := s0`pc WITH [(P) := sPopL11],
                              aux_seq := d]
          ELSE s1 = s0 WITH [pc := s0`pc WITH [(P) := sPopLLoopTop]]
          ENDIF)

  tPopL12_15(s0, a, s1): bool =
      (∃ P:
         a = popL12_15(P) ∧
          s0`pc(P) = sPopL11 ∧
           LET R = reference_class_assign(s0`heap, s0`lh(P),
                               record_assign(reference_class_select(s0`heap, s0`lh(P)),
                                                  "L", s0`DUMMY))
             IN s1 = s0 WITH [heap := R, pc := s0`pc WITH [(P) := sIdle]])
 END snark
```

G.2 deseq.pvs

```
deseq[T: TYPE]: THEORY
 BEGIN

  t: VAR T

  between(i: int, j: int): TYPE = {x: int | i < x ∧ x < j}

  Deseq_Base: NONEMPTY_TYPE =
  [# left: int, right: int, seq: [between(left, right) → T] #]

  db: VAR Deseq_Base

  Deseq: NONEMPTY_TYPE = {db | db`right > db`left}

  d: VAR Deseq

  length(d): nat = d`right − d`left − 1

  empty?(d): bool = length(d) = 0

  NonemptyDeseq: TYPE = {d | ¬ empty?(d)}

  ned: VAR NonemptyDeseq

  EMPTY_DESEQ: Deseq =
      (# left := 0,
         right := 1,
         seq := λ (i: between(0, 1)): ε! (t: T): TRUE #)

  push_left(d, t): NonemptyDeseq =
```

```
    (# left := d`left - 1,
       right := d`right,
       seq := λ (i: between(d`left - 1, d`right)):
              IF i = d`left THEN t ELSE d`seq(i) ENDIF #)

 pop_left(ned): [Deseq, T] =
    ((# left := ned`left + 1,
        right := ned`right,
        seq := λ (i: between(ned`left + 1, ned`right)): ned`seq(i) #),
     ned`seq(ned`left + 1))

 push_right(d, t): NonemptyDeseq =
    (# left := d`left,
       right := d`right + 1,
       seq := λ (i: between(d`left, d`right + 1)):
              IF i = d`right THEN t ELSE d`seq(i) ENDIF #)

 pop_right(ned): [Deseq, T] =
    ((# left := ned`left,
        right := ned`right - 1,
        seq := λ (i: between(ned`left, ned`right - 1)): ned`seq(i) #),
     ned`seq(ned`right - 1))
END deseq
```