A Proof Framework for Concurrent Programs

Leonard Lensink, Sjaak Smetsers and Marko van Eekelen

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
Email: {l.lensink,s.smetsers,m.vaneekelen}@cs.ru.nl

Abstract. This paper presents a proof framework for verifying concurrent programs that communicate using global variables. The approach is geared towards verification of models that have an unbounded state size and are as close to the original code as possible. The bakery algorithm is used as a demonstration of the framework basics, while the (full) framework with thread synchronization was used to verify and correct the reentrant readers writers algorithm as used in the Qt library.

Keywords: Formal Verification, Synchronization, Concurrency, PVS, SPIN.

1 Introduction

Parallelism has been employed for many years, mainly in high-performance computing. The physical constraints preventing an unlimited growth in processor speed have led to a revival of interest in concurrent computing. One can observe that parallel computing has become a dominant paradigm in computer architecture, especially for multi-core systems [12].

Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs. In practice, it can be incredibly difficult to track down these software bugs, because of their unpredictable nature: they are typically caused by infrequent 'race conditions' that are hard to reproduce. In such cases, it is necessary to thoroughly investigate 'suspicious' parts of the system in order to improve these components in such a way that correctness is guaranteed. The most thorough technique is to formally verify properties of the system under investigation.

In [13] a case study is presented that combines two formal verification methods, namely model checking and theorem proving. The idea is to use the model checker to track down and remove (concurrency) bugs, and to use the theorem prover to formally prove their absence. Model checkers and theorem provers have their own input languages. Therefore, the use of these formal tools requires that the original program is first converted to (modeled in) the language of the model checker, and subsequently translated into the language of the theorem prover. Obviously, both translations introduce potential sources of errors, particularly if these translations are performed manually.

The experience with this case study led us to develop a *general proof framework* with specific support to construct proofs of mutual exclusion, deadlock and starvation properties for concurrent algorithms that communicate using shared variables. The proof framework consists of a *model* that can be instantiated and used for different programs, a *set of theorems* that can be used to prove relevant properties of

the system and a *general approach* towards solving the proofs and proof obligations generated by the framework. Using SPIN [5] as model checker, we investigate how (concurrent) Promela (the input language of SPIN) programs can be modeled in PVS [9]. We define an automatic translation within the framework that serves as a basis not only to facilitate the conversion of Promela into PVS, but also to support the investigation of general properties of parallel computer programs.

In this paper, this framework is introduced. The use of the framework is explained applying it to a common mutual exclusion algorithm known as the *bakery* algorithm [6]. We demonstrate the power of the framework by applying it to a larger example, showing correctness of a solution to the reentrant readers-writers problem [14] that improves upon the widely used Qt C++ library by Trolltech. In that paper [14] it was described how a model was constructed and checked using the SPIN model checker. This revealed an error in the implementation, and a correction was suggested. The improved algorithm was subsequently shown to be correct in PVS. However, the PVS model was constructed manually. Here, we show how the model can be generated automatically, and how the proof can be structured using the support of the framework.

Section 2 introduces the framework basics. In Section 3 these framework basics are applied to a classic example, the bakery algorithm. Section 4 adds thread synchronisation to the framework and applies it to a large example, the reentrant readers-writers problem. Section 5 draws conclusions and suggests future work.

2 Framework basics

The general approach is to take a piece of (parallel) code, and model it in a model checker to detect bugs. Subsequently, after improving the model it will be subject to verification in a theorem prover. To do this systematically, an embedding of the semantics of the model checker in the theorem prover is required. In our case, we use PVS as theorem prover and Promela as the modeling language. The embedding presented in this paper is a mixed shallow/deep one: the framework is based on a shallow embedding while the translation of the model into PVS exploits the native features of PVS as much as possible, and hence can be seen as a deep embedding.

2.1 State transition system

In essence, a SPIN model is a state transition system with temporal logic. Our framework reflects this directly by representing concurrently executing threads by means of a state transition system. Each process runs in a *thread*. The semantics of executing threads are captured in a theory that specifies that each thread is either *Running*, *Waiting*, or *Blocked*. All threads have a threadid `tid` of type `TID`, a program counter `pc`, a return address `rtn` and a local store `local`. The types of these entities are provided as theory type parameters, and will be supplied by the concrete (translated) Promela program. The theory parameter `NT` denotes the number of concurrently executing threads.

```
Threads[NT:posnat, PC, LS, GS: TYPE] : THEORY BEGIN
  TID     : TYPE = below(NT)[1]
  TStatus : TYPE = { Running, Waiting, Blocked }
```

[1] Denotes the set of natural numbers between 0 and `NT`, exclusive of `NT`.

```
  TState  : TYPE = [# tid: TID, status: TStatus, local:LS, pc, rtn: PC #][2]
  Threads : TYPE = [ TID → TState]
  System  : TYPE = [# threads: Threads, current: TID, global: GS #]
  currThread(s: System): TState = s`threads(s`current)[3]
END Threads
```

The entire system state consists of all the threads combined with the global variable store `global` (again a theory type parameter), and a variable `current` signifying which thread is currently executing. The utility function `currThread` yields the state of the currently executing thread.

The (global) state transition relation of the system is determined by the (local) state transition of the concurrently executing threads, which is specified by means of a `step` relation. This relation is defined in a separate theory `Model`, also containing definitions of the entities required by `Threads`. This `Model` theory, resulting from the translation of the concrete Promela program (say P), has the following skeleton.

```
Model[NT:posnat] : THEORY BEGIN
  PC:TYPE= below(  ... the size of P ...  )
  GV:TYPE= [#  global variables appearing in P  #]
  LV:TYPE= [#  local variables of each thread in P  #]

  IMPORTING Threads[NT, PC, LV, GV]

  step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV): bool
    =  generated instructions from P
```

The effect of `step` is local, i.e. it only influences the local state of the currently executing thread, and possibly the global state of the system. The local states of other threads are unaffected, which also follows from `step`'s type. To enforce this kind of locality for the entire system, we introduce the parameterized predicate `PredSys` on `System` that when applied to a system s, identifies all valid predecessors of s.

```
PredSys(s: System): pred[System] =
  { p: System | ∀(ot:TID): ot≠s`current ⇒ p`threads(ot) = s`threads(ot) }
```

As usual, we will model parallel execution by non-deterministic interleaving. To anticipate on the proving process we already include the notion of invariancy, by means of an predicate on the `System` type, called `invSystem`. This leads to the following `interleave` relation, performing one execution step of a randomly selected running thread.

```
interleave(ps:(invSystem)[4], ss: System) : bool =
  PredSys(ss)(ps) ∧  currThread(ps)`status = Running ∧
  LET cs=ps WITH [`current = ss`current][5] IN
    step(currThread(cs),cs`global)(currThread(ss),ss`global)
```

[2] Recordtypes in PVS are surrounded by `[#` and `#]`.
[3] `r`x` denotes the selection of the `x`-component of record `r`.
[4] PVS allows predicates to be used as types.
[5] `r WITH [`x := e]` denotes a new record that is equal to `r` except for the `x`-component which has value `e`.

Theorems and proofs

PVS has no innate notion of deadlock or starvation, so these have to be defined explicitly. *Deadlock states* are usually defined as states for which there are no outgoing edges. Although *final states* may have no outgoing edges, they are not considered as deadlock states. Assume that `zeroState` denotes a predicate identifying these final states, we can formulate *deadlock-freeness* as:

```
∀(s1:(invSystem)): ¬zeroState(s1) ⇒ ∃(s2:System): interleave(s1,s2)
```

This interpretation of deadlock is often not precise enough. Consider for example a situation where a process executes a non terminating loop (because it is waiting for something that will never occur). Then, it might be that all other threads are waiting for this one to terminate before they can proceed. According to the definition there would be no deadlock. To handle this situation a refined notion of deadlock-freeness is needed. This refinement is based on the observation that if there exists a (well-founded) order $<$ on states such that from every non-final state s of the system a transition can be made to a state t with $t < s$, then the system will be free of deadlock. More formally:

```
NoDeadlock(s:(invSystem)) : bool =
    ¬zeroState(s) ⇒ ∃(t:System): interleave(s, t) ∧ t < s
```

Proving deadlock-freeness of a system boils down to giving an appropriate state ordering and showing that the generated `step` relation indeed has this `NoDeadlock` property.

The previous theorem can also guarantee freedom from starvation, if fairness of scheduling is imposed on the system. However, most (efficient) thread implementations do not provide this way of scheduling. Therefore, we introduce a more sophisticated notion of starvation freedom that makes no specific assumptions on the scheduling regimen. We base this notion on the following intuition: if a process *intends* to perform a certain action it will *eventually* be able to. The intention is signaled by a process entering a certain execution path. For instance, executing the instruction that puts it on the path to enter a critical section. Execution of the intended action is signaled by reaching the goal instruction, e.g. if the process finally gets the permission to enter the critical section. This leads to the following definitions, in which both intention and goal are specified as `PC` values.

```
NoStarvation(s:(invSystem), t:TID, enter, goal:PC) : bool =
 s`threads(t)`PC = enter ⇒ eventually(s, t, goal)

eventually(s1:(invSystem), t:TID, goal:PC): RECURSIVE bool = ∀(s2:System):
  interleave(s1,s2) ⇒ s2`threads(t)`pc = goal ∨ eventually(s2,t,goal)
MEASURE noStarvationMeasure(s1,t)
```

In PVS all functions must be total. For recursive functions, such as `eventually` above, a so called *measure* must be provided. This measure, based on some well-founded order, ensures that at least one of the function arguments strictly decreases (according to the order) at each recursive call, thus ensuring termination. In the case above, termination also guarantees freedom of starvation, because only a finite number of interleaving steps are possible before the thread reaches its goal. Proving the absence of starvation boils down to giving a proper definition

of `noStarvationMeasure`. In combination with deadlock-freeness this gives that eventually the goal will be reached. In the sequel, we will also specify the state ordering for deadlock-freeness as a measure with the obvious name `noDeadlockMeasure`.

In general, these measures will involve parts of the global system state as well as properties of individual threads. In order to define and facilitate reasoning about these measures the following small PVS theory proves to be very useful. It contains a folding operation `fsum` that accumulates the results of a function `fun`, provided as a parameter. The lemma relates the results of `fsum` applied to *comparable* functions f and g (i.e. there exists at most one argument for which f and g produce different results).

```
fsum(m:upto(N),fun:[below(N)→nat]):RECURSIVE nat =
  IF m=0 THEN 0 ELSE fun(m-1)+fsum(m-1,fun) ENDIF MEASURE m

fsum_diff: LEMMA
  ∀(n:upto(N),k:below(n),f,g:[below(N)→nat]):
    (∀(m:below(n)): m≠k ⇒ f(m)=g(m)) ⇒ fsum(n,f)+g(k) = fsum(n,g)+f(k)
```

2.2 Constructing invariants

For all properties to be proven, it needs to be established that the invariant used as a precondition for `interleave` is maintained throughout all the transitions. This property directly follows from a similar property for the `step` relation that resulted from the translation of the original program into our framework. The exact nature of the invariant therefore also depends on the model that is being verified. Usually, it is very hard to invent the right invariant before conducting the proof. Therefore, it is convenient to start with a minimal (weak) pre-condition which is gradually strengthened while the proof is constructed. Unfortunately, proofs are usually quite brittle and redevelopment of the proofs due to changed premises can be very time-consuming. To facilitate the iterative development of proofs, the invariant will be structured as follows:

```
invSystem1: pred[System] = λ(s:System): Prop1(s) ∧ invSystem2(s)
invSystem2: pred[System] = λ(s:System): Prop2(s) ∧ invSystem3(s)
...
invSystemN: pred[System] = λ(s:System): true
```

Each part of the invariant $\texttt{invSystem}_i$ consists of a single property $\texttt{Prop}_i$ and the next part of the invariant $\texttt{invSystem}_{i+1}$. Proving is conducted in a breadth-first manner. If the current invariant appears to be too weak, the proving process is interrupted, and the invariant is extended with a new property. Restarting the proof and redoing the proof steps that where done during the previous session is now easy, because the added extensions do not interfere with the steps that were done before. One can directly proceed to the place where the proof was interrupted, and continue the proof process, most probably by expanding the current invariant in order to use the newly added property.

2.3 Translating Spin models to the framework

In this section we show how Promela programs are translated into our PVS framework. Since we focus on concurrent systems in which processes communicate via shared variables, it is not necessary to cover Promela completely. In particular,

the inter process communication via channels is left out. The core of the translation is the way Promela statements are treated. To facilitate a formal presentation, we introduce an abstract syntax for Promela statements that serve as input to the translator. As a result, we do not generate PVS directly, but make use of an intermediate target language *IL* which can be converted almost directly into an appropriate PVS theory. This is done to keep the core translation simple: some peephole optimizations, in particular small transformations that reduce the state space, can now be performed in a separate phase. The conversion from *IL* to PVS is not fully elaborated but informally exemplified.

The syntax of Abstract Promela Statements is given in the left column of the table below. $\overrightarrow{s}$ denotes 0 or more and $<s>$ denotes 0 or 1 occurrences of s.

$\mathbb{L}$	: $x, y, \ldots$	local variables
$\mathbb{G}$	: $X, Y, \ldots$	global variables
V	::= $\mathbb{L} \mid \mathbb{G}$	all variables
$\mathbb{P}$	: $p, q, \ldots$	procedure names
E_{int}	: $1, x + y, \ldots$	integer expressions
E_{bool}	: $\mathsf{true}, x > 3, \ldots$	boolean expressions
E	::= $E_{\mathsf{int}} \mid E_{\mathsf{bool}}$	all expressions
SM	: $\mathsf{LOCK}, \mathsf{UNLOCK}, \mathsf{WAIT}, \mathsf{TRANS}, \mathsf{NOTIFY}$	synchronization operations

PS	::= $V <[E_{\mathsf{int}}]> := E$	IL	::= $\mathsf{ASS}\ V <[E_{\mathsf{int}}]>\ E$
	$\mid$ $\mathsf{if}\ \overrightarrow{G}\ <\mathsf{else}\ TE>\ \mathsf{fi}$		$\mid$ $\mathsf{GOTO}\ PC$
	$\mid$ $\mathsf{do}\ \overrightarrow{G}\ <\mathsf{else}\ TE>\ \mathsf{od}$		$\mid$ $\mathsf{SWITCH}\ \overrightarrow{(E_{\mathsf{bool}}, PC)}\ <PC>$
	$\mid$ $\mathbb{P}$		$\mid$ $\mathsf{CALL}\ PC$
	$\mid$ $\mathsf{atomic}\ \overrightarrow{PS}$		$\mid$ RTN
	$\mid$ $G.SM$		$\mid$ ATO
G	::= $E_{\mathsf{bool}} \rightarrow TE$		$\mid$ OTA
TE	::= $\langle \overrightarrow{PS}, bool \rangle$		$\mid$ $SM\ LI$
		PC	::= $\mathbb{N}$
		LI	::= $\mathbb{N}$

The abstract syntax (PS) reflects the essential statements of Promela: assignments, choices, and repetitions. The left-hand side of an assignment can be either a simple variable of the element of an array, explaining the optional selection. The boolean in the then or else statement (TE) indicates whether or not the corresponding sequence of statements ends with a break. Functions in Promela are inlined. However, to reduce the size of generated code, we refrain from inlining and use simple procedure calls (no parameters, no result) instead. The synchronization operations (indicated by SM in the grammar), are not part of standard Promela. They are explained in Section 4. Note that (boolean and integer) expressions are not specified further; we can almost directly interpret these as PVS code.

The intermediate language given in the right column is largely self-explanatory. It has been designed in such a way that, on the one hand, it completely covers the intended source language, and, on the other hand, it can be interpreted directly by means of an appropriate PVS theory. *IL* resembles traditional low-level assembly languages, with the exception of the SWITCH instruction used in the translation of both choices and repetitions. This instruction takes a sequence of (boolean) expression-address pairs and randomly chooses one of the addresses corresponding

to expressions evaluating to true. If none of the mentioned expressions is true, then either the else address is chosen (if available), or the instruction will block. The chosen address will become the new program counter value of the currently executing process.

We will describe the treatment of statements only; the translation of a complete model including procedure definitions, and local and global variable declarations is straightforward. The translation of an (abstract) Promela statement into the intermediate language *IL* is defined by the following set of mutual recursive functions $s[\![\cdot]\!]_\rho$. Here ρ is an environment mapping function names to PC values.

$$
\begin{aligned}
PS[\![v := e]\!]_\rho\ pc &= (pc+1, [\mathsf{ASS}\ v\ e]) \\
PS[\![\mathsf{if}\ gs\ eo\ \mathsf{fi}]\!]_\rho\ pc &= (pc_e, [\mathsf{SWITCH}\ gl\ el] +\!\!+\ il_g +\!\!+\ il_e)\ \mathit{where} \\
&\quad (pc_g, gl, il_g) = \overrightarrow{G}[\![g]\!]_\rho\ pc+1\ pc_e\ pc_e \\
&\quad (pc_e, el, il_e) = <TE>[\![e]\!]_\rho\ pc_g\ pc_e\ pc_e \\
PS[\![\mathsf{do}\ gs\ eo\ \mathsf{od}]\!]_\rho\ pc &= (pc_e, [\mathsf{SWITCH}\ gl\ el] +\!\!+\ il_g +\!\!+\ il_e)\ \mathit{where} \\
&\quad (pc_g, gl, il_g) = \overrightarrow{G}[\![g]\!]_\rho\ pc+1\ pc\ pc_e \\
&\quad (pc_e, el, il_e) = <TE>[\![e]\!]_\rho\ pc_g\ pc\ pc_e \\
PS[\![p]\!]_\rho\ pc &= (pc+1, [\mathsf{CALL}\ \rho(p)\ pc+1]) \\
PS[\![\mathsf{atomic}\ s]\!]_\rho\ pc &= (pc'+1, [\mathsf{ATO}] +\!\!+\ il +\!\!+\ [\mathsf{OTA}])\ \mathit{where} \\
&\quad (pc', il) = \overrightarrow{PS}[\![s]\!]_\rho\ pc+1
\end{aligned}
$$

$$
\begin{aligned}
\overrightarrow{PS}[\![[\ ]]\!]_\rho\ pc &= (pc, [\ ]) \\
\overrightarrow{PS}[\![s : ss]\!]_\rho\ pc &= (pc'', il +\!\!+\ il')\ \mathit{where} \\
&\quad (pc', il) = PS[\![s]\!]_\rho\ pc \\
&\quad (pc'', il') = \overrightarrow{PS}[\![ss]\!]_\rho\ pc'
\end{aligned}
$$

$$
\begin{aligned}
\overrightarrow{G}[\![[\ ]]\!]_\rho\ pc\ c\ e &= (pc, [\ ], [\ ]) \\
\overrightarrow{G}[\![b \to s : gs]\!]_\rho\ pc\ c\ e &= (pc'', (b, l) : gl', il +\!\!+\ il')\ \mathit{where} \\
&\quad (pc', l, il) = TE[\![s]\!]_\rho\ pc\ c\ e \\
&\quad (pc'', gl, il') = \overrightarrow{G}[\![gs]\!]_\rho\ pc'\ c\ e
\end{aligned}
$$

$$
\begin{aligned}
<TE>[\![\Diamond]\!]_\rho\ pc\ c\ e &= (pc, \Diamond, [\ ]) \\
<TE>[\![<e>]\!]_\rho\ pc\ c\ e &= (pc', <el>, il)\ \mathit{where} \\
&\quad (pc', el, il) = TE[\![e]\!]_\rho\ pc\ c\ e
\end{aligned}
$$

$$
\begin{aligned}
TE[\![\langle [\ ], b\rangle]\!]_\rho\ pc\ c\ e &= (pc, l, [\ ])\ \mathit{where} \\
&\quad l = \mathit{if}\ b\ \mathit{then}\ e\ \mathit{else}\ c \\
TE[\![\langle ss, b\rangle]\!]_\rho\ pc\ c\ e &= (pc'+1, pc, il +\!\!+\ [\mathsf{GOTO}\ l])\ \mathit{where} \\
&\quad (pc', il) = \overrightarrow{PS}[\![ss]\!]_\rho\ pc \\
&\quad l = \mathit{if}\ b\ \mathit{then}\ e\ \mathit{else}\ c
\end{aligned}
$$

3 An example: bakery algorithm

As an example we apply our method to Lamports bakery algorithm: a classic solution to the problem of N-mutual exclusion. The algorithm itself is analogue to the way bakeries give their customers turns by drawing a number from a machine, where the baker serves the lowest number when he is available. The algorithm listed

below as a sequence of *PS* statements is essentially the same as Lamport's original. The translation of the program to *IL* is given below on the right-hand side.

```
Enter[tid] := true;                0  ASS Enter[tid] true
h := 0;                            1  ASS h 0
i := 0;                            2  ASS i 0
do i<NT                            3  SWITCH (i<NT,4) 9
 -> if h>Num[i]                    4  SWITCH (h>Num[i],5) 7
      -> h := Num[i];              5  ASS h Num[i]
       else ; fi;                  6  GOTO 7
    i := i + 1;                    7  ASS i i + 1
   else break;
od;                                8  GOTO 3
Num[tid] := h + 1;                 9  ASS Num[tid] h + 1
Enter[tid] := false;               10 ASS Enter[tid] false
i := 0;                            11 ASS i 0
do i<NT && !Enter[i]               12 SWITCH(i<NT&&!Enter[i],13)(i>=NT,16)
 -> if Num[i]=0->;                 13 SWITCH(Num[i]=0,14)
       Num[i]>Num[tid]->;               (Num[i]>Num[tid],14)
       Num[i]=Num[tid]&&i>=tid->;       (Num[i]=Num[tid]&&i>=tid,14)
    fi;
    i := i + 1;                    14 ASS i i + 1
 i >= NT -> break;
od;                                15 GOTO 12
Num[tid] := 0;                     16 ASS Num[tid] 0
```

The complete model will execute the above code infinitely many times. In Spin, it is impossible to model check this example, because the drawn numbers are unbounded leading to an infinite state space. There exist several versions of the algorithm that use finite numbers when drawing, see Section 5.

Next we translate this *IL* program into the PVS framework. To reduce the number of different states of our model some of the statements are combined. Particularly, multiple assignments are implemented by a single instruction if they contain at most one (read/write) access to a global variable. For instance, the first three assignments of our example are combined into a single transition. For the implementation of the SWITCH statements, we will use COND expressions of PVS.

This yields the following instantiation of the `Model` skeleton. This theory also contains the proper instances of the parameters of `Threads` from Section 2. The step relation is not fully specified. For brevity only characteristic cases of this relation are given.

```
Model[NT:posnat] : THEORY BEGIN
  PC:TYPE= below(11)
  GV:TYPE= [# enter:ARRAY[below(NT) → boolean],num:ARRAY[below(NT) → nat] #]
  LV:TYPE= [# h: nat, i: nat #]
  IMPORTING Threads[NT, PC, LV, GV]

  step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV): bool = LET pc=lv1`pc IN
  COND
    pc=0 → lv2=lv1 WITH [`local`h := 0,`local`i := 0,`pc := 1] ∧
           gv2=gv1 WITH [`enter(lv1`tid) := TRUE],
    pc=1 → COND lv1`local`i<NT → lv2=lv1 WITH [`pc := 2 ],
           ELSE → lv2=lv1 WITH [`pc := 5 ] ENDCOND∧ gv2=gv1,
```

```
    ...
    pc=5 → lv2=lv1 WITH [`pc := 6 ] ∧
           gv2=gv1 WITH [`num(lv1`tid) := lv1`local`h + 1],
    ...
    pc=10 → lv2=lv1 WITH [`pc := 0 ] ∧ gv2=gv1 WITH [`num(lv1`tid) := 0]
  ENDCOND
END Model
```

Theorems and proofs

Proving the different properties requires (1) the instantiation of the (`noDeadlockMeasure` and `noStarvationMeasure`) measures needed for the theorems defined in Section 2.1, and (2) the definition of an invariant strong enough to prove that these measures indeed strictly decrease.

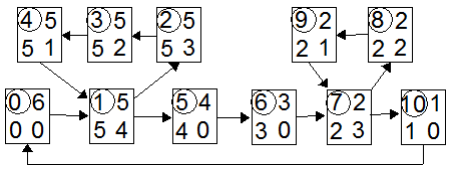
As to (1), we can observe the following:

- The states themselves can be given a numerical ordering where each state in the control flow has a smaller number, with the starting state the smallest.
- If there is no transition possible to a smaller state according to the above numerical ordering, there is an increase of the local variable `i` until the maximum value of `NT` is reached.

This gives the following measure, defined using the `fsum` function.

```
NoDeadlockMeasure(s:System): [nat,nat,nat] = (fsum(NT,mapBL(s)),
    fsum(NT,λ(t:TID):NT-s`threads(t)`local`i), fsum(NT,mapBR(s)))
```

The ordering that is used is the lexicographical ordering on 3-tuples. The two auxiliary functions simply map the value of the program counter of a thread to a natural number. The `mapBR` values for each state are shown in the bottom right-hand corner and the `mapBL` in the bottom left-hand corner of the corresponding box (see the diagram below). The encircled numbers in the upper left-hand corner correspond to the value of the program counter. The fourth value (in the upper right-hand corner), given by `mapUR`, is used further on.



Absence of starvation means that if a process *intends* to enter the critical section, it will eventually do so. This is formulated using the program counter. Once a process has obtained a number it arrives at the state with program counter is 6, from where it will proceed to the critical section at location 10.

```
Enter: PC = 6
Goal:  PC = 10
BakeryStarvationFree: THEOREM
   ∀ (s:(InvSystem), t:TID): NoStarvation(s, t, Enter, Goal)
```

In order to prove this, we define the starvation measure based on the following system properties:

- Let $peers_t$ denote the set of processes that were choosing just after thread t has received its number.
- A thread t that draws a number will always get a larger one, except for the members of $peers_t$.
- For each thread t the size of $peers_t$ will only get smaller.
- The set of drawn numbers that are in front of the process that wants to enter the critical section, will only get smaller.
- It is possible to map the program counter to a natural number in such a way that these numbers will get smaller or the local variable `i` will decrease.

To keep track of peers we extend the state of each thread with a *ghost/model* variable named `peers`. This value of `peers` is set to the value of the global variable `enter` at the moment the thread has drawn its number (the location with program counter 5), and reset to an empty set after leaving the critical section (indicated by the program counter value 10). This leads to some small modifications of the `step` relation, where `peers` is set at location 5 and 10:

```
step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV): bool =
  LET pc=lv1‘pc IN
  COND ...
    pc=5 → lv2=lv1 WITH [‘pc := 6, peers = gv1‘enter ]
         ∧ gv2=gv1 WITH [‘num(lv1‘tid) := lv1‘local‘h + 1],
    ...
    pc=10 → lv2=lv1 WITH [‘pc := 0, peers = ∅ ]
         ∧ gv2=gv1 WITH [‘num(lv1‘tid) := 0]
```

The `starvationMeasure` introduced in Section 2 can now be defined as follows. The corresponding ordering is the lexicographical ordering on 5-tuples.

```
b2N(b:bool): nat = IF b THEN 1 ELSE 0 ENDIF

starvationMeasure(s:System,t:TID):[nat,nat,nat,nat,nat]
= ( fsum(NT, b2N o s‘threads(t)‘peers)
  , fsum(NT, λ(t2:TID): LET nr = s‘global‘num IN
       b2N (nr(t2) ≠ 0∧ (nr(t2),t2)<(nr(t),t)∧ ¬s‘threads(t)‘peers(t2)))
  , fsum(NT, mapUR(s))
  , fsum(NT, λ(t2:TID): NT-s‘threads(t2)‘local‘i)
  , fsum(NT, mapBR(s)) )
```

An interesting safety property of our system is, of course, *mutual exclusion*: it should be impossible for two processes to be in the critical section at the same time. More concretely, when a process has 10 as its program counter, it will be the only one.

```
inCS(s:System,t:TID): bool = s‘threads(t)‘pc = 10

MutualExclusion(s:System) : bool =
  ∀(t1:TID): inCS(s,t1) ⇒ ∀(t2:TID): inCS(s,t2) ⇒ t1=t2
```

Before proving this property, we first explain some of our program invariant definitions. At the beginning of the proof process it may not be entirely clear what invariants will be needed. Therefore, these invariants are progressively constructed as explained on page 5.

The transition relationship defined in `step` generates type correctness conditions. For instance, when the `num` array is indexed, the index may not exceed the total

number of processes. This leads to the following property of the first invariant of the system:

```
numAccessed(pc:PC): bool = pc = 2 ∨ pc = 3 ∨ pc = 7 ∨ pc = 8

Prop1(s:System): bool =
  ∀(t:TID): numAccessed(s`threads(t)`pc) ⇒ s`threads(t)`local`i < NT
```

The most important invariant stipulates that whenever a process is in the loop where it compares the numbers drawn by each thread (indicated by the predicate `comparing`), then for all threads it has already examined, the current thread is greater according to the lexicographical ordering on `(num(t),t)`. In the same part of the program execution it also holds that if a thread is in the peer group, it cannot have the `enter` flag set. This is expressed by the property of the second invariant.

```
comparing(pc:PC): bool = pc = 7 ∨ pc = 8 ∨ pc = 9

Prop2(s:System): bool = ∀(t:TID):
  LET ts=s`threads(t) IN comparing(ts`pc) ⇒
    ∀ (k:TID): k ≠ t ∧ (k < ts`local`i ∨ k = ts`local`i ∧ ts`pc = 9) ⇒
      (s`global`num(t),t) < (s`global`num(k),k) ∧
      ts`peers(k) ⇒ ¬ s`global`enter(k))
```

All that has to be established further is that if a process enters the comparison loop, it will do so only if it has a number greater than all the numbers already given out. The only exception is made for processes that are in the peer group. In order to prove this, we need some extra invariants that are pretty straightforward. Their PVS code is left out for brevity.

- Processes can be entering only in states 0,1,2,3,4, and 5.
- After setting the peer group, each process is always part of its own peer group.
- At the beginning (states 0,1,2,3, and 4) the peer group is empty. At these states also the `num` value is 0; otherwise greater than 0.
- Finally, in state 10, `i` is always equal to `NT`.

The invariants guarantee that when a process proceeds to the critical section (location 10), all the other processes have larger numbers. This enables the proof of the measures. The safety property also follows directly from the invariant combined with the fact that the lexicographical ordering is well founded and has only one smallest element. The proofs of the theorems proceed by a case distinction on the value of the program counter, creating a symbolic execution of the algorithm. For all the possible cases only instances of the `fsum_diff` lemma (Section 2) and the invariant are needed to discharge all the proof goals. The simple structure of the proofs makes it feasible to prove larger algorithms, like the reentrant readers writers algorithm given in the next section, although their proofs end up being quite large. The proof file for the latter program is more than 20,000 lines.

4 Framework with thread synchronisation

Many concurrent algorithms are based on locking primitives that modern operating systems usually support. These primitives are not available in standard Promela but added to the framework. In principle we could have modeled these locking primitives in Promela (like the bakery algorithm) and translated this model to PVS

using the procedure as described in the previous sections. However, it appears to be more convenient to extend Promela with special synchronization constructs (In fact, we've already anticipated on this extension in the definition of the abstract Promela syntax; see Section 2.3), and use a shallow embedding by incorporating basic locks also into our PVS framework.

4.1 Incorporating locking primitives

The idea of the basic locks is similar to, for example, the synchronization mechanism of Java. Shared resources are protected by locks. If a process wants exclusive access to these resources it performs a *lock* operation on the corresponding lock. Releasing a resource is done by calling *unlock*. Besides, processes should be able to relinquish their turn temporarily by means of a *wait* command and also be able to wake other processes up using *notify*. Another primitive is *transfer*, which allows the process to explicitly hand over the execution privilege to the first waiting process. This operation plays an essential role in our algorithm in order to guarantee absence of starvation. Furthermore, we have built in basic support for implementing atomic statements. In Promela one can enforce a sequence of statements to be non-interruptible by placing these statements in an atomic context. Although these atomic statements can be simulated in PVS by locks, we prefer to represent them more efficiently by a separate system extension. It suffices to use a single global boolean to indicate whether the currently executing process is interruptible. This leads to the following adapted `Threads` theory.

```
Threads[NT, NL:posnat, PC, LS, GS:TYPE] : THEORY BEGIN
  TID : TYPE = below(NT)
  LID : TYPE = below(NL)
  TState, Threads:  TYPE /* as before */
  LState: TYPE = [# lockedBy: lift[TID],  blocked, waiting: list[TID] #]
  Locks : TYPE = [ LID → LState]
  System: TYPE = [# threads:Threads,locks:Locks,
                    atomic:bool,current:TID,global:CV #]
END Threads
```

The new theory parameter `NL` denotes the number of locks appearing in the program, also used to identify each lock by a `LID`. This also explains why the lock variables of our intermediate language *IL* were represented by natural numbers; see Section 2.3. The system state now contains a variable `locks` holding the `LState` of each lock. This state indicates whether the lock is occupied (in which case `lockedBy` refers to the corresponding thread) and maintains two queues for holding the blocked and waiting processes. The boolean variable `atomic` indicates that no context switch is allowed. The lock operations are defined as a separate PVS theory. As an example the implementation of the `transfer` operation is given.

```
LOCK[NT, NL:posnat,PC:TYPE,LV:TYPE,GV:TYPE]: THEORY BEGIN
IMPORTING Threads[NT, NL, PC, LV, GV]
LSystem(lid:LID): TYPE = { s: System | s`locks(lid)`lockedBy = up(s`current) }

lock  (lid:LID)(s:System):       System
unlock(lid:LID)(s:LSystem(lid)): System

transfer(lid:LID)(s:{ s1: LSystem(lid) | cons?(s1`locks(lid)`waiting) } ):
```

```
  LSystem(car(s`locks(lid)`waiting)) = LET ls = s`locks(lid) IN
    s WITH [`threads(car(ls`waiting))`status := Running,
            `locks(lid)`lockedBy  := up(car(ls`waiting)),
            `locks(lid)`waiting   := cdr(ls`waiting)]

wait  (lid:LID)(s:LSystem(lid)): System
notify(lid:LID)(s:LSystem(lid)): System
END LOCK
```

As usual, a process can only perform an unlock, wait, transfer or notify if it is the owner of the lock. This requirement is expressed in the dependent type `LSystem`. Moreover, transfer has the additional requirement that it is only allowed if the waiting queue of the corresponding lock is not empty. Again, this is enforced by defining the type of the system parameter dependently.

In our framework, a thread can only access its own state and the global variables of the system; see the `step` relation. However, a thread executing a synchronization operation may indirectly affect other system components. It may even change the status of other threads. Instead of passing the complete system state to the (local) step relation, we have implemented these 'system calls' by extending the result of `step` with a function of type `[System → System]`. This yields the adjusted types of `step`, `sysStep` and `interleave`:

```
step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV,sc:[System → System]): bool

sysStep(s1: (invSystem), s3:System):bool= ∃(s2:System, sc:[System → System]):
  step(currThread(s1),s1`global)(currThread(s2),s2`global,sc) ∧ s3 = sc(s2)

interleave(s1:(invSystem),s2:System):bool= PredSys(s2)(s1) ∧
 LET ct=s2`current IN
  IF s1`atomic THEN ct=s1`current ∧ sysStep(s1,s2)
  ELSE s1`threads(ct)`status=Running ∧ sysStep(s1 WITH [`current:=ct],s2)
  ENDIF
```

4.2 Example: reentrant read-write

A more complex synchronization mechanism involves processes that acquire access to resources for both reading and writing: the classic readers-writers problem. Several kinds of solutions exist. Here, we will consider a reentrant *read-write* locking mechanism that employs writers preference. A thread can acquire the lock multiple times, even when the thread has not fully released the lock: locking can be *reentrant*. Most solutions give priority to write locks over read locks because write locks are assumed to be more important, smaller, exclusive, and occurring less frequently. The main disadvantage of this choice is that it can result in *reader starvation*: when there is always a thread waiting to acquire a write lock, threads waiting for a read lock will never be able to proceed.

Specifying the entire algorithm would take too much space. The part that shows the Promela version of *readLock* used for acquiring the lock for reading is given below. As one can see, the locks appearing in this program are represented by variable names. In our translation these names will be mapped to natural numbers. This is not included in the translation function, but can be added straightforwardly (e.g. by parameterizing the translation with an additional environment that performs this mapping). The result of the translation is on the right-hand side of the listing.

```
Mutex.LOCK;                        0 LOCK 0
if Count[tid]=0 ->                 1 SWITCH (Count[tid]=0,2) 9
 do CurrWr!=NT||WaitWr> 0 ->       2 SWITCH (CurrWr!=NT||WaitWr>0,3) 7
      WaitRe := WaitRe + 1;        3 ASS WaitRe (WaitRe + 1)
      Mutex.WAIT;                  4 WAIT 0
      WaitRe := WaitRe - 1;        5 ASS WaitRe (WaitRe - 1)
    else break;
 od;                               6 GOTO 2
 ThrCount := ThrCount + 1;         7 ASS ThrCount (ThrCount + 1)
else ; fi;                         8 GOTO 9
Count[tid] := Count[tid] + 1       9 ASS Count[tid] (Count[tid] + 1)
Mutex.UNLOCK;                      10 UNLOCK 0
                                   11 RTN
```

The part of the `step` relation that corresponds to this program fragment is shown below. In the complete model, the values of the program counter depend on the exact location of this function in the original program, which may be different from the given values.

```
Model[NT:posnat] : THEORY BEGIN
PC : TYPE = below(8)
GV: TYPE = [# count:ARRAY[below(NT)→nat],CurrWr,WaitWr,WaitRe,ThrCount:nat #]
LV: TYPE = [# rNest, wNest, maxLocks: nat #]
step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV,sc:SysCall): bool =
  LET pc = lv1`pc IN
  COND
    pc=0 → lv2 = lv1 WITH [`pc := 1] ∧ gv1 = gv2 ∧ sc = lock(0),
    pc=1 → COND gv1`count(lv1`tid)=0 → lv2 = lv1 WITH [`pc := 2],
            ELSE → lv2 = lv1 WITH [`pc := 6] ENDCOND ∧ gv1 = gv2 ∧ sc=id,
    ...
    pc=7 → lv2 = lv1 WITH [`pc := lv1`rtn] ∧ gv2 = gv1 ∧ sc=id
  ENDCOND
END Model
```

The complete model also contains a few ghost variables (`rNest`, `wNest` and `maxLocks`) that limit the number of nested locks a process is allowed to use. If no such limit was imposed, it would be impossible so show absence of starvation. The `noDeadlockMeasure` consists of the sum of all local `maxLocks` variables, the sum of all global counts, and again a mapping from program counters to natural numbers similar to the one used in the bakery algorithm. Absence of deadlocks, follows directly from the lexicographical ordering on these 3-tuples.

The fact that a process that wants to obtain a writers lock (and has ended up in the wait queue), will finally obtain one, requires a starvation measure that tracks the position of that particular thread in the waiting queue. It will either shift a position in the queue or other processes will use up their allotted locks, limited by the `maxLock` counter. For this measure, the mapping of program counters to natural numbers is slightly more complicated because these values also depend on the status of the thread.

Een reviewer wil graag weten hoe de optization waarover we het hebben compared met andere approaches. Ik weet daar niet echt een antwoord op.

The invariant needed to prove the theorems is large, but revolves around the relationships of the possible values of the variables used in the program at certain points in their execution past, similar to what was done in Section **??**. The PVS model that was used in the concrete proof was adjusted in order to reduce the

number of possible state transitions. This manually performed optimization was based on the observation that if a model uses a single lock and all accesses to global variables are synchronized (which is the case in our example) one can use the atomic instead of the (first) lock of the system. This means that a process will never have status `Blocked`. The code for the `wait`, `notify` and `transfer` needs to be adjusted in order to obtain the correct behavior[6].

5 Related work

Providing support for domain specific theorem proving environments within general theorem provers in the area of state transition systems is present in TAME [1]. However, this tool set offers tactics and templates to construct proofs using PVS and is geared towards proving properties of SCR, timed and I/O automata.

Basten and Hooman [3] provide an indirect approach to proof support for models that originate from model checkers. They first define the semantics of process algebra in PVS and then investigate the difference in proving behavior depending of the kind of embedding that is used.

For the purpose of developing consistent requirement specifications, [4] introduces a framework that is used for the transformation of transition systems (given as specifications in the model checker Uppaal [7]) to specifications in PVS.

A deep embedding of Promela lite (a Promela like language) is given in [11]. However, they use this embedding to prove lemmas concerning symmetry detection and not to prove properties of specific models.

For finite state models, translating from the model checker to the theorem prover can be circumvented by using the PVS built-in model checker [8].

In [10] model checking and theorem proving are combined to analyze the classic readers-writers problem. However, the authors start from a tabular specification of the solution rather than from a real algorithm. This tabular specification is translated straightforwardly into SPIN and PVS. Some properties (like safety and clean completion) can be derived semi-automatically.

The bakery algorithm is a classical solution to the mutual exclusion problem. In Lamport's original version the numbers drawn by the customers can grow infinitely, leading to an unlimited state space which makes it unsuited for being model checked directly. However, several modifications have been proposed to restrict the drawn numbers also leading to a finite state space, e.g. see [2]. The advantage of using a theorem prover is, of course, that there are no limitations on the values being used. This made it possible to work directly with the original unbounded algorithm.

6 Conclusions and future work

In this paper we have presented a framework for constructing formal correctness proofs of PROMELA models. The framework is restricted to concurrent processes that communicate via global variables. It enables reasoning about basic synchronization protocols (such as the bakery algorithm) as well as more complex synchronization mechanisms (such as the reentrant read/write locks provided by the Qt library). The framework provides basic theories and proof support for constructing proofs of fundamental concurrency properties, such as (absence of) deadlock and

[6] The full PVS files of both examples can be found at `http://www.cs.ru.nl/S.Smetsers/frameworkexamples`.

starvation. Formulating these properties is structured by introducing suitable abstract functions and predicates that are instantiated based on the original model. Proving actually boils down to constructing an appropriate invariant and to showing that this invariant indeed holds for the constructed state transition relation.

Our future plans are to extend the framework in such a way that it covers the complete PROMELA language, e.g. by adding constructs for modeling message passing. Furthermore, the proof process can be partially automated by defining appropriate PVS-tactics to avoid repeating certain sequences of proof steps. Finally, many auxiliary mappings of program counters to natural numbers that were needed to define proper measures, can be generated automatically.

References

1. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 2000.
2. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
3. Twan Basten and Jozef Hooman. Process Algebra in PVS. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS, vol. 1579, pp 270-284, 1999.
4. Adriaan de Groot. *Practical Automaton Proofs in PVS*. PhD thesis, Radboud University Nijmegen, 2008.
5. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
6. L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, August 1974.
7. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
8. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *Computer-Aided Verification (CAV)*. LNCS, vol. 1102, pp. 411-414, 1996.
9. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Conf. on Automated Deduction (CADE)*. LNAI, vol. 607, pp. 747-752, 1992.
10. Vera Pantelic, Xiao-Hui Jin, Mark Lawford, and David Lorge Parnas. Inspection of concurrent systems: Combining tables, theorem proving and model checking. In *Software Engineering Research and Practice*. pp. 629-635, 2006.
11. Shamim Ripon and Alice Miller. Verification of symmetry detection using pvs. *ECEASST*, 35, 2010.
12. Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.
13. B. van Gastel, L. Lensink, S. Smetsers, and M. van Eekelen. Reentrant Readers-Writers. In *Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS)*. LNCS, vol. 5596, pp. 85-102, 2009.
14. B. van Gastel, L. Lensink, S. Smetsers, and M. van Eekelen. Deadlock and Starvation Free Reentrant Readers-Writers. *Sci. Comput. Program.*, 76(2):82–99, 2011.

The following appendices give the complete code of which parts appear in the paper. This is meant to make reviewing more easy and this code is not intended to be part of the final paper. The paper should be readable without referencing these appendices.

A Step function bakery

```
Model[NT:posnat] : THEORY
BEGIN     IMPORTING Threads[NT, PC, LV, GV]
  PC: TYPE = below(11)
  GV: TYPE = [# enter: ARRAY [below(NT) → boolean]
               , num: ARRAY [below(NT) → nat] #]
  LV: TYPE = [# h: nat, i: nat #]

  step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV): bool = LET pc=lv1`pc IN
  COND
    pc=0 → lv2=lv1 WITH [`local`h := 0,`local`i := 0,`pc := 1]
           ∧ gv2=gv1 WITH [`enter(lv1`tid) := TRUE],
    pc=1 → COND lv1`local`i<NT → lv2=lv1 WITH [`pc := 2 ],
                ELSE → lv2=lv1 WITH [`pc := 5 ]
           ENDCOND∧ gv2=gv1,
    pc=2 → COND lv1`local`h>gv1`num(lv1`local`i) →
                    lv2=lv1 WITH [`pc := 3 ],
                ELSE →  lv2=lv1 WITH [`pc := 4 ]
           ENDCOND∧ gv2=gv1,
    pc=3 → lv2=lv1 WITH [`local`h := gv1`num(lv1`local`i),`pc := 4 ]
           ∧ gv2=gv1,
    pc=4 → lv2=lv1 WITH [`local`i := lv1`local`i + 1,`pc := 1 ]
           ∧ gv2=gv1,
    pc=5 → lv2=lv1 WITH [`pc := 6 ]
           ∧ gv2=gv1 WITH [`num(lv1`tid) := lv1`local`h + 1],
    pc=6 → lv2=lv1 WITH [`local`i := 0,`pc := 7]
           ∧ gv2=gv1 WITH [`enter(lv1`tid) := FALSE],
    pc=7 → COND lv1`local`i<NT ∧  ¬gv1`enter(lv1`tid) →
                    lv2=lv1 WITH [`pc := 8 ],
                lv1`local`i≥NT →  lv2=lv1 WITH [`pc := 10 ],
                ELSE → FALSE
           ENDCOND∧ gv2=gv1,
    pc=8 → COND gv1`num(lv1`local`i)=0 OR
                gv1`num(lv1`local`i)>gv1`num(lv1`tid) OR
                gv1`num(lv1`local`i)=gv1`num(lv1`tid) ∧
                lv1`local`i≥lv1`tid →
                   lv2=lv1 WITH [`pc := 9],
                ELSE → FALSE
           ENDCOND∧ gv2=gv1,
    pc=9 → lv2=lv1 WITH [`local`i := lv1`local`i + 1,`pc := 7 ]
           ∧ gv2=gv1,
   pc=10 → lv2=lv1 WITH [`pc := 0 ]
           ∧ gv2=gv1 WITH [`num(lv1`tid) := 0]
END Model
```

B Lock implementation PVS

```
LOCK[NT, NL:posnat,PC:TYPE,LV:TYPE,GV:TYPE]: THEORY
BEGIN   IMPORTING Threads[NT, NL, PC, LV, GV]
SysCall : TYPE = [System → System]
LSystem(lid:LID): TYPE = { s: System | s`locks(lid)`lockedBy = up(s`current) }

lock(lid:LID)(s: System): System =
  LET ts = s`threads(s`current),
      ls = s`locks(lid) IN
  IF up?(ls`lockedBy)
  THEN s WITH [`threads(s`current)`status := Blocked,
               `locks(lid)`blocked := append(ls`blocked,cons(s`current, null))]
  ELSE s WITH [`locks(lid)`lockedBy := up(s`current)]
  ENDIF

unlock(lid:LID)(s: LSystem(lid)): System =
  LET ls = s`locks(lid) IN
  IF null?(ls`blocked)
  THEN s WITH [`locks(lid)`lockedBy := bottom ]
  ELSE s WITH [`threads(car(ls`blocked))`status := Running,
               `locks(lid)`lockedBy  := up(car(ls`blocked)),
               `locks(lid)`blocked   := cdr(ls`blocked) ]
  ENDIF

wait(lid:LID)(s: LSystem(lid)): System =
  LET sn = s WITH [`threads(s`current)`status := Waiting,
                   `locks(lid)`waiting := append(s`locks(lid)`waiting
                                                 ,cons(s`current, null))]
  IN unlock(lid)(sn)

transfer(lid:LID)(s: { s1: LSystem(lid) | cons?(s1`locks(lid)`waiting) } ):
    LSystem(car(s`locks(lid)`waiting)) =
  LET ls = s`locks(lid) IN
        s WITH [`threads(car(ls`waiting))`status := Running,
            `locks(lid)`lockedBy  := up(car(ls`waiting)),
            `locks(lid)`waiting   := cdr(ls`waiting)]

notify(lid:LID)(s:LSystem(lid)): System =
  LET ls = s`locks(lid) IN
  s WITH [`locks(lid)`blocked   := append(ls`blocked,ls`waiting),
          `locks(lid)`waiting   := null,
          `threads := lambda(t:TID):
                        IF member(t,ls`waiting)
                        THEN s`threads(t) WITH [`status := Blocked ]
                        ELSE s`threads(t) ENDIF ]

startAtom(s: { s1: System | ¬ s1`atomic }): { s1: System | s1`atomic } =
  s WITH [`atomic := TRUE ]

endAtom(s: { s1: System | s1`atomic}): { s1: System | ¬ s1`atomic } =
  s WITH [`atomic := FALSE ]
END LOCK
```

C Step function readerswriters

```
Model[NT:posnat] : THEORY
BEGIN   IMPORTING LOCK[NT, 1, PC, LV, GV]

PC : TYPE = below(12)

GV: TYPE = [# count: ARRAY [below(NT) → nat],
              CurrentWriter, WaitingWriters,
              WaitingReaders, ThreadCount: nat #]
LV: TYPE = [# readNest, writeNest, maxLocks: nat]

step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV,sc:SysCall): bool =
  LET pc = lv1`pc IN
  COND
    pc=0 → lv2 = lv1 WITH [`pc := 1] ∧ gv1 = gv2 ∧ sc = lock(0),
    pc=1 → COND gv1`count(lv1`tid)=0 → lv2 = lv1 WITH [`pc := 2],
                ELSE → lv2 = lv1 WITH [`pc := 6]
            ENDCOND ∧ gv1 = gv2 ∧ sc=id,
    pc=2 → COND gv1`CurrentWriter ≠ NT ∨ gv1`WaitingWriters > 0
                    → lv2 = lv1 WITH [`pc := 3],
                ELSE → lv2 = lv1 WITH [`pc := 5]
            ENDCOND ∧ gv1 = gv2 ∧ sc=id,
    pc=3 → lv2 = lv1 WITH [`pc := 4] ∧
            gv2 = gv1 WITH [`WaitingReaders := gv1`WaitingReaders + 1] ∧
            sc=wait(0),
    pc=4 → lv2 = lv1 WITH [`pc := 2] ∧
            gv2 = gv1 WITH [`WaitingReaders := gv1`WaitingReaders - 1] ∧
            sc=id,
    pc=5 → lv2 = lv1 WITH [`pc := 6] ∧
            gv2 = gv1 WITH [`ThreadCount := gv1`ThreadCount + 1] ∧
            sc=id,
    pc=6 → lv2 = lv1 WITH [`pc := 7] ∧
            gv2 = gv1 WITH [`count(lv1`tid) := gv1`count(lv1`tid) + 1] ∧
            sc=unlock(0),
    pc=7 → lv2 = lv1 WITH [`pc := lv1`rtn] ∧ gv2 = gv1 ∧ sc=id
NDCOND
END Model
```