Model-checking lock-sharing systems with tree automata

Corto Mascle, Anca Muscholl and Igor Walukiewicz

CONCUR 2023

Lock-sharing systems

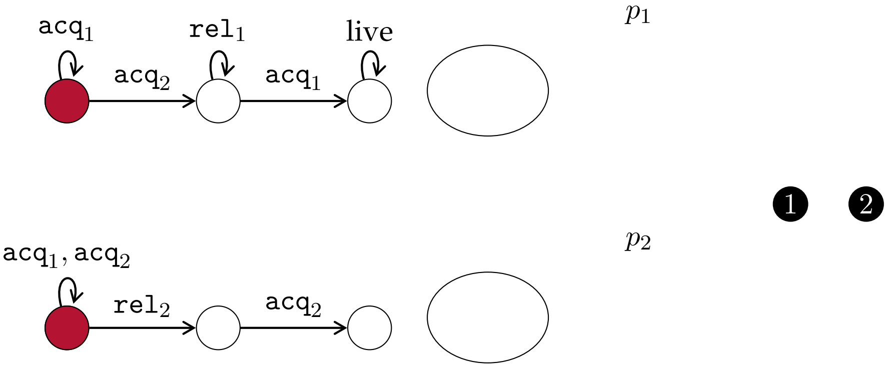
Lock-sharing system [Kahlon, Ivancic, Gupta '05]

$Proc$: set of processes
$Locks$: set of locks.

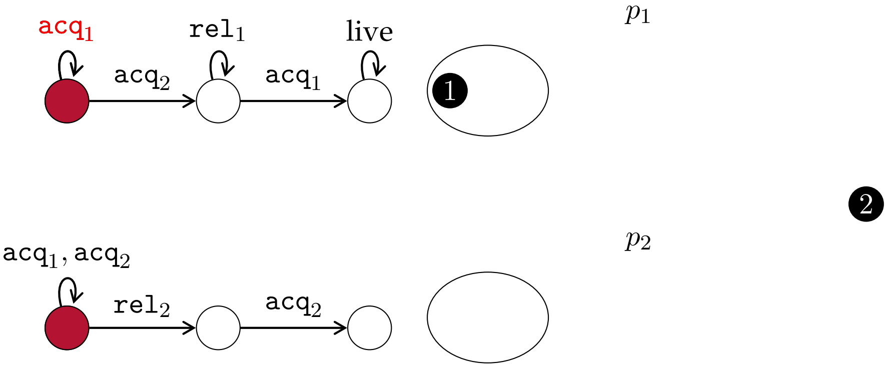
Lock-sharing system (LSS):
$\mathcal{A}_p = (S_p, \Sigma_p, \delta_p, init_p)$ for each $p \in Proc$.

Transitions include operations on locks :
$\delta_p : S_p \times \Sigma_p \rightarrow Op_T \times S_p$ with $Op_T = \{acq_t, rel_t \mid t \in T\} \cup \{nop\}$.

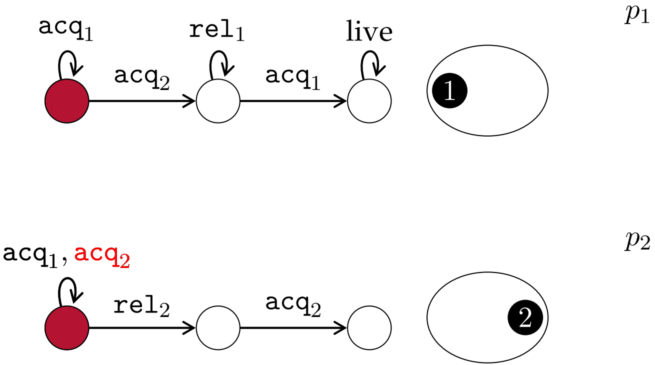
Semantics



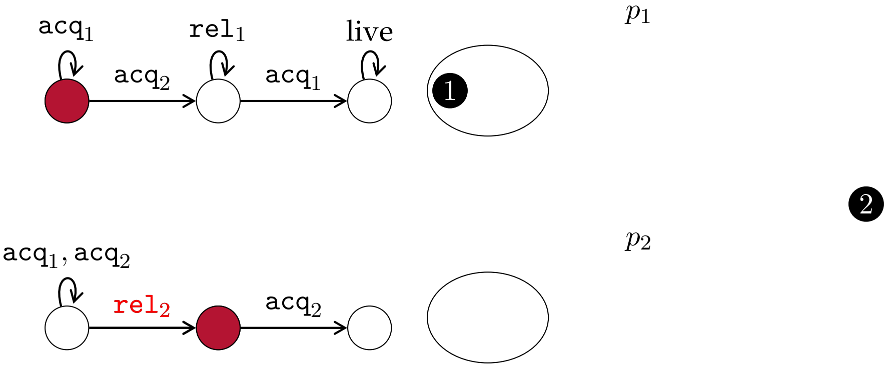
Semantics



Semantics

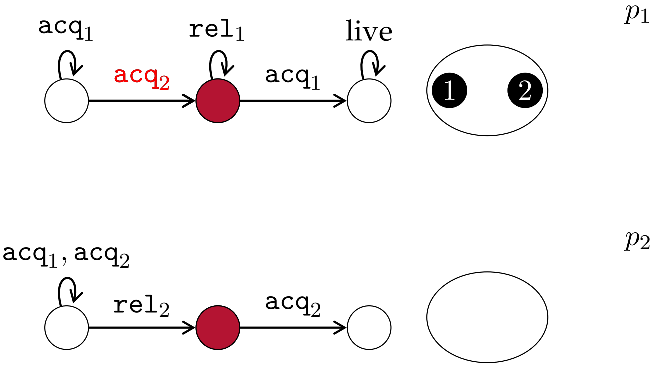


Semantics

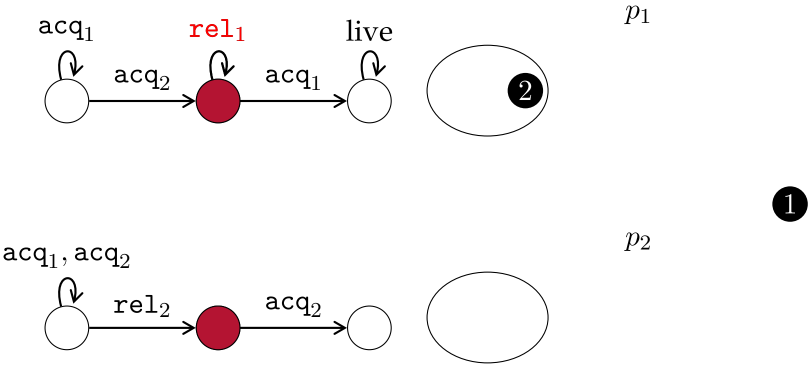


p_1
$\mathtt{acq}_1$
$\mathtt{rel}_1$
live
$\mathtt{acq}_2$
$\mathtt{acq}_1$
1
2
p_2
$\mathtt{acq}_1, \mathtt{acq}_2$
$\mathtt{rel}_2$
$\mathtt{acq}_2$

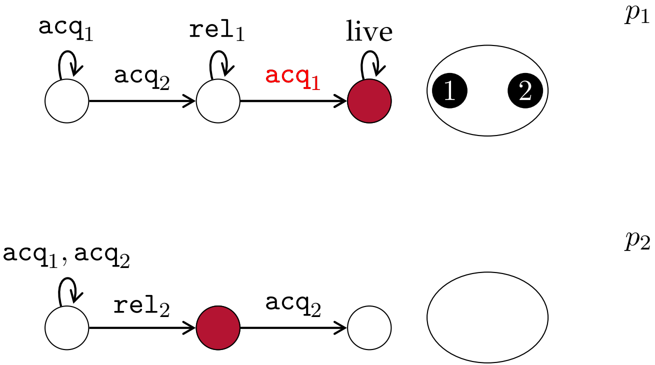
Semantics



Semantics



Semantics



Locks vs Variables

With variables (and atomic read-write)

▷ Processes can synchronize completely.

▷ Verification is hard (PSPACE for finite-state)

Locks vs Variables

With variables (and atomic read-write)

▹ Processes can synchronize completely.

▹ Verification is hard (PSPACE for finite-state)

With locks

▹ No way to test if a lock is taken $\rightarrow$ Lock $<$ Boolean variable

Locks vs Variables

With variables (and atomic read-write)

▹ Processes can synchronize completely.

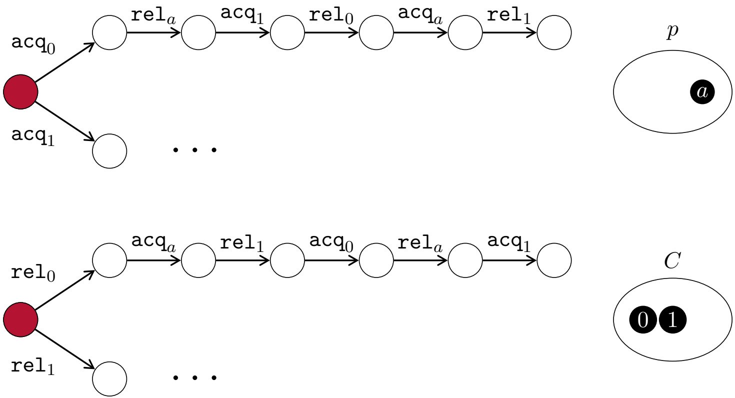
▹ Verification is hard (PSPACE for finite-state)

With locks

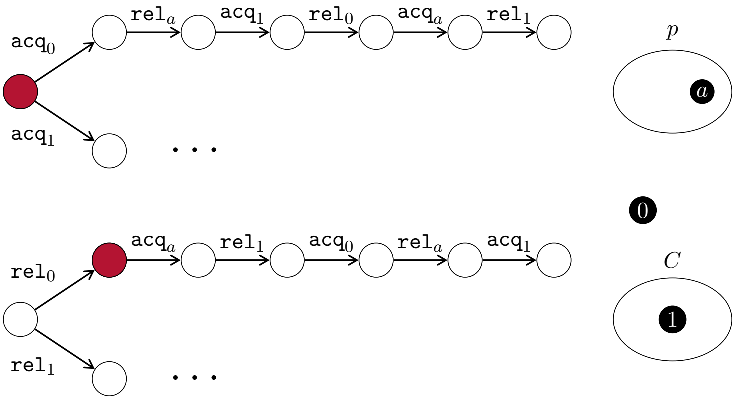
▹ No way to test if a lock is taken $\rightarrow$ Lock < Boolean variable

▹ Variables can be simulated by interleaving lock acquisitions

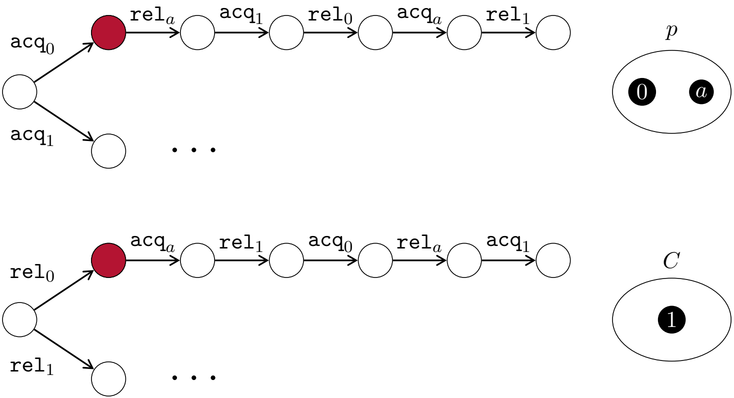
Passing information



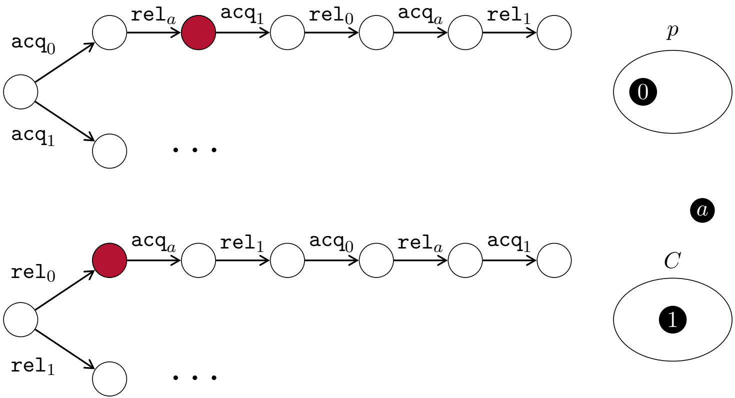
Passing information



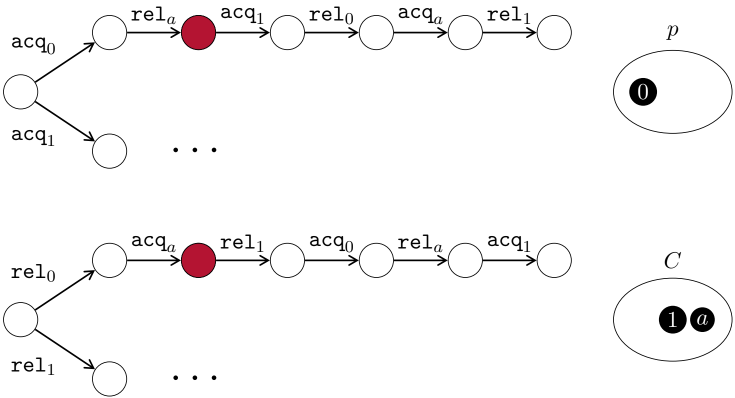
Passing information



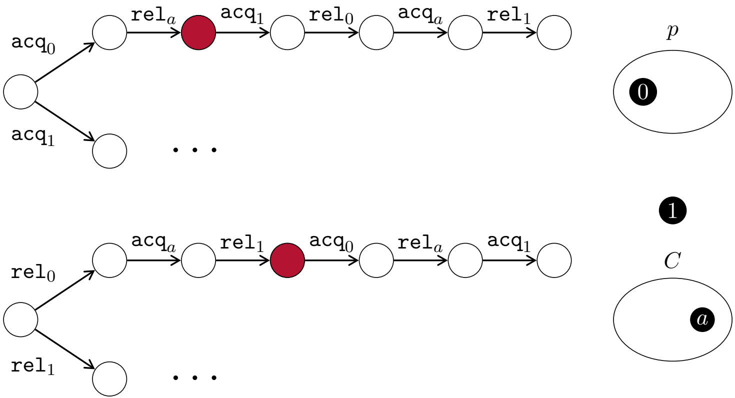
Passing information



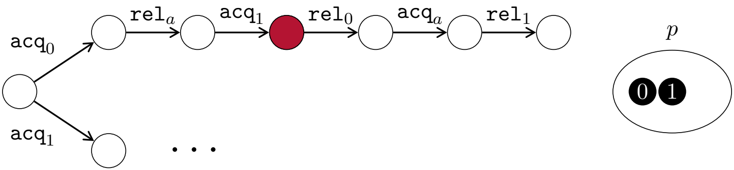
Passing information

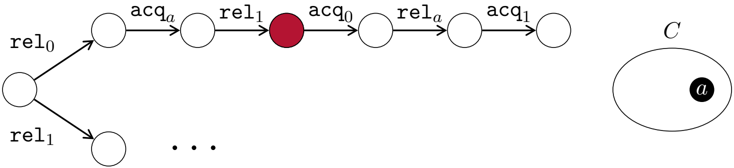


Passing information

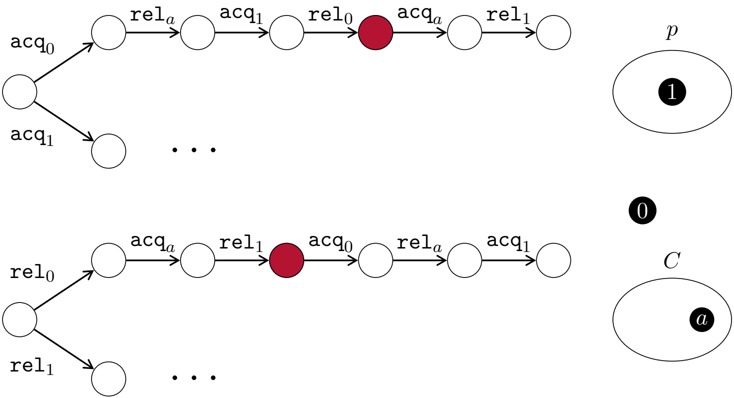


Passing information

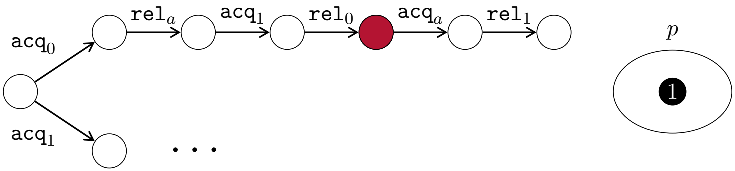


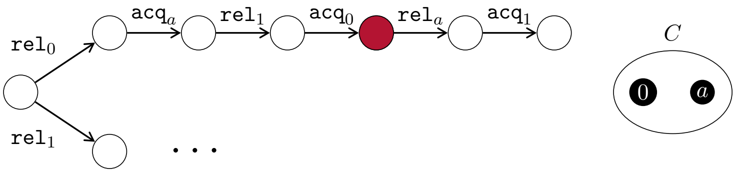


Passing information

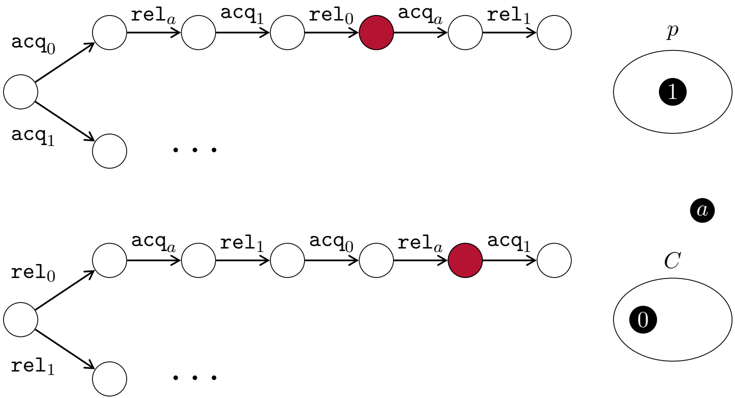


Passing information

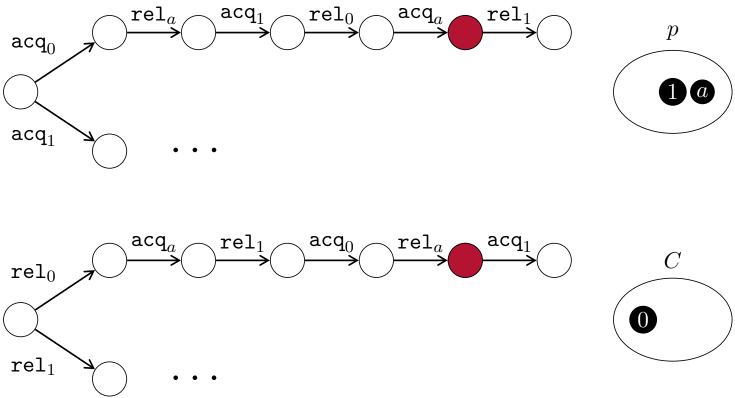




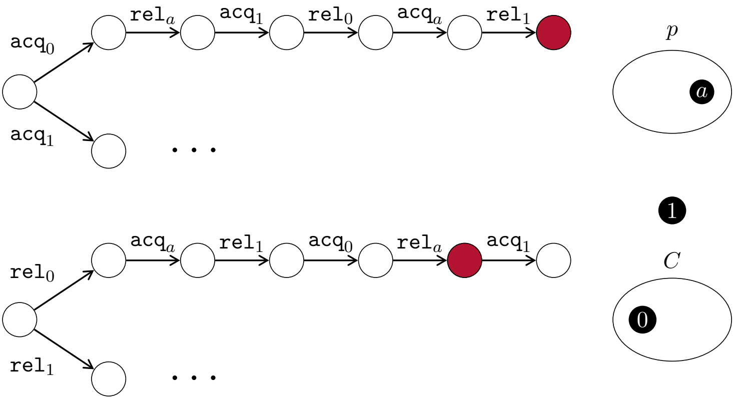
Passing information



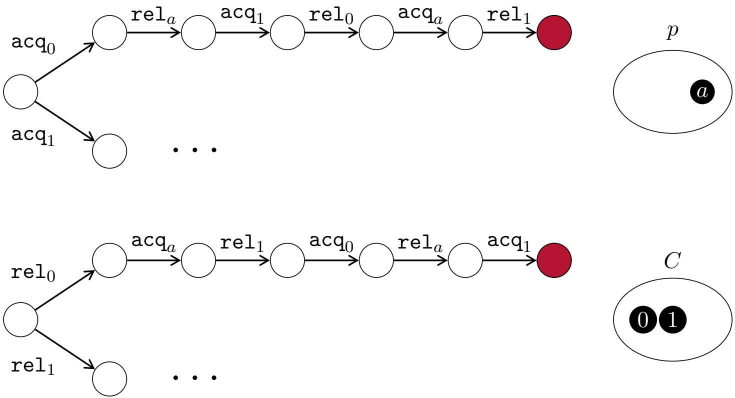
Passing information



Passing information

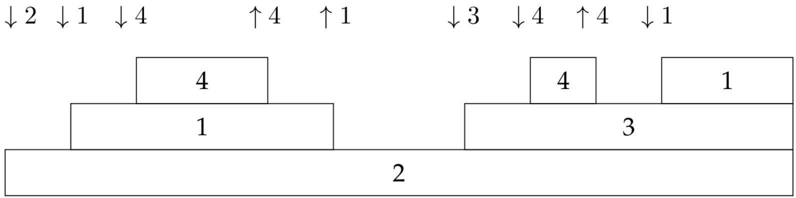


Passing information



Nested locking

All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



Now we cannot simulate variables!

Dynamic LSS

▷ We want to allow an unbounded amount of processes and locks.

Dynamic LSS

- ▷ We want to allow an unbounded amount of processes and locks.

- ▷ Processes will now be able to spawn other processes

Dynamic LSS

▷ We want to allow an unbounded amount of processes and locks.

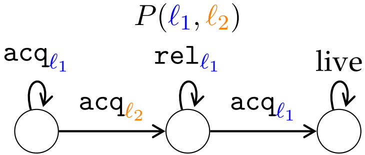
▷ Processes will now be able to spawn other processes

▷ A process now takes parameters, represented by *lock variables*

$$Proc = \{P(\ell_1, \ell_2), Q(\ell_1, \ell_2, \ell_3), R(), ...\}$$

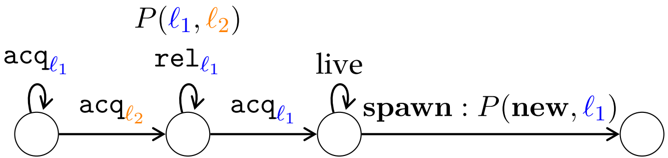
Dynamic LSS

$Locks :$

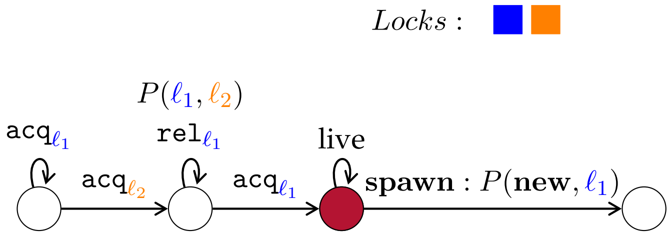


Dynamic LSS

$Locks :$

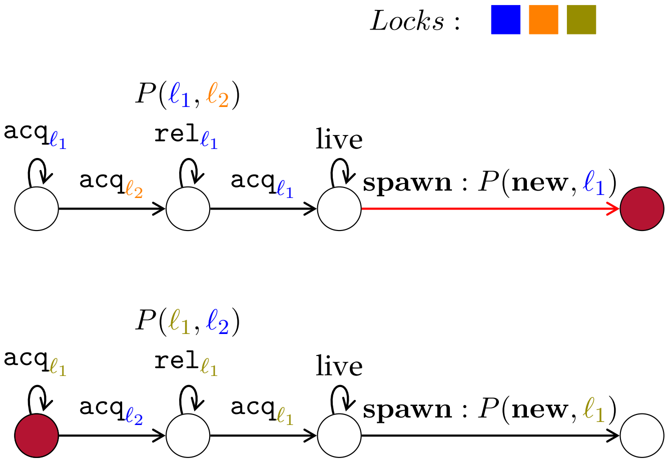


Dynamic LSS



$Locks:$
$P(\ell_1, \ell_2)$
$\texttt{acq}_{\ell_1}$
$\texttt{acq}_{\ell_2}$
$\texttt{rel}_{\ell_1}$
$\texttt{acq}_{\ell_1}$
live
$\textbf{spawn} : P(\textbf{new}, \ell_1)$

Dynamic LSS

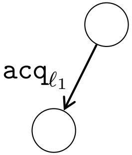

$Locks:$
$P(\ell_1, \ell_2)$
$\texttt{acq}_{\ell_1}$
$\texttt{rel}_{\ell_1}$
live
$\texttt{acq}_{\ell_2}$
$\texttt{acq}_{\ell_1}$
$\textbf{spawn} : P(\textbf{new}, \ell_1)$
$P(\ell_1, \ell_2)$
$\texttt{acq}_{\ell_1}$
$\texttt{rel}_{\ell_1}$
live
$\texttt{acq}_{\ell_2}$
$\texttt{acq}_{\ell_1}$
$\textbf{spawn} : P(\textbf{new}, \ell_1)$

Tree representation

We represent finite or infinite runs by trees

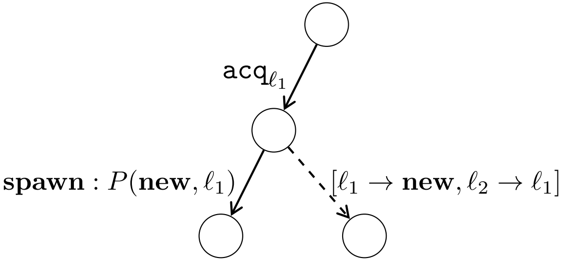
Tree representation

We represent finite or infinite runs by trees



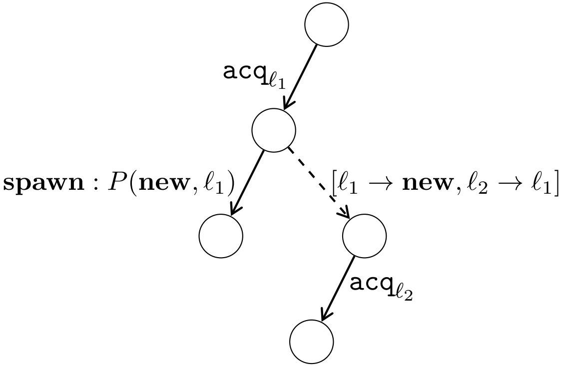
Tree representation

We represent finite or infinite runs by trees



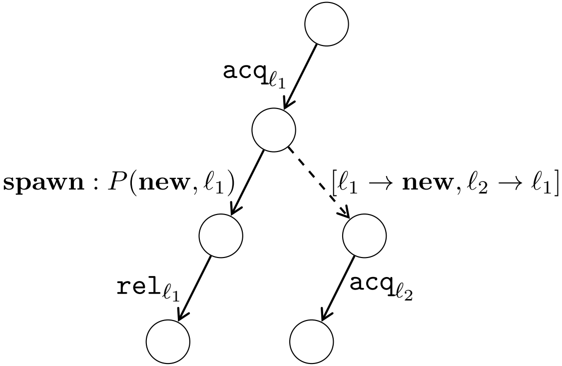
Tree representation

We represent finite or infinite runs by trees



Tree representation

We represent finite or infinite runs by trees



Tree specifications

We assume runs to be *fair*: If a process can execute a step infinitely many times, it eventually does.
Deadlock $\Leftrightarrow$ finite tree.

Labels

We label each node of the tree with the asymptotic behaviour of the locks associated with its variables in the subtree.

- $AG\ \neg\ell_i \rightarrow$ never taken in the subtree

Labels

We label each node of the tree with the asymptotic behaviour of the locks associated with its variables in the subtree.

- $AG\ \neg\ell_i \rightarrow$ never taken in the subtree
- $G\ \ell_i \rightarrow$ never released

Labels

We label each node of the tree with the asymptotic behaviour of the locks associated with its variables in the subtree.

- $AG \neg\ell_i \rightarrow$ never taken in the subtree
- $G\ \ell_i \rightarrow$ never released
- $EFG\ \ell_i \rightarrow$ taken at some point and never released

Labels

We label each node of the tree with the asymptotic behaviour of the locks associated with its variables in the subtree.

- $AG\ \neg\ell_i \rightarrow$ never taken in the subtree
- $G\ \ell_i \rightarrow$ never released
- $EFG\ \ell_i \rightarrow$ taken at some point and never released
- $AFG\ \neg\ell_i \rightarrow$ held finitely many times

Labels

We label each node of the tree with the asymptotic behaviour of the locks associated with its variables in the subtree.

- $AG\ \neg\ell_i \rightarrow$ never taken in the subtree
- $G\ \ell_i \rightarrow$ never released
- $EFG\ \ell_i \rightarrow$ taken at some point and never released
- $AFG\ \neg\ell_i \rightarrow$ held finitely many times
- $AGF\ \neg\ell_i \rightarrow$ always released but may be taken infinitely many times

Labels

We label each node of the tree with the asymptotic behaviour of the locks associated with its variables in the subtree.

- $AG\ \neg\ell_i \rightarrow$ never taken in the subtree
- $G\ \ell_i \rightarrow$ never released
- $EFG\ \ell_i \rightarrow$ taken at some point and never released
- $AFG\ \neg\ell_i \rightarrow$ held finitely many times
- $AGF\ \neg\ell_i \rightarrow$ always released but may be taken infinitely many times

Lemma

Consistency of those labels can be checked by an exponential Büchi automaton.

Order on locks

We also label nodes with *local orders*.
$\ell_1 \preceq \ell_2$ if ℓ_2 is taken after ℓ_1 was taken and never released.

Lemma

A tree is *schedulable* if it can be enriched with **consistent labels** and **consistent acyclic local orders**.

Result

Lemma

There exists an **exponential Büchi tree automaton** recognising realizable run trees of DLSS.
It is **polynomial** if the number of locks is fixed.

Theorem

Model-checking DLSS against regular tree specifications is EXPTIME-complete.

Right-resetting pushdown tree automata

Right-resetting = the stack is emptied every time we go to a right child.

Lemma

Emptiness is Fixed-parameter tractable for right-resetting parity pushdown tree automata.

Theorem

Model-checking pushdown DLSS against regular tree specifications is EXPTIME-complete.

Future work

Add variables ?

$\rightarrow$ Easy VASS encoding

$\rightarrow$ Find good restrictions on variables to make the problem more tractable.

Thank you for your attention!