Nested Atomic Sections with Thread Escape: A Formal Definition

Frédéric Dabrowski, Frédéric Loulergue, **Thomas Pinsard**

LIFO - Université d' Orléans

18/11/2013

Journées GDR LTP

Concurrent programming

- Multi-core trend
- Gap between programming languages and hardware
- No suitable abstraction to deal with shared data

Locks : the most common pattern

- Map operations to protect to a lock
- Responsibility to the **user** to prevent interference
- Error prone : deadlocks, need to understand the whole program
- Dilemma of the granularity : coarse/inefficient or fine/complex

Example

```
count := 0; Lock l;
procedure inc_count(){
  lock(l)
  {
    count := count+1;
  }
  unlock(l)
}
procedure main()
{
  a := fork(inc_count);
  b := fork(inc_count);
  print(count);
}
```

Atomic sections : an alternative ?

- Responsibility to the **run-time system** or the **compiler** to prevent interference
- The user only delimits the region to protect

Example

```
count := 0;
procedure inc_count(){
  atomic
  {
    count := count+1;
  }
}
```

```
procedure main()
{
  a := fork(inc_count);
  b := fork(inc_count);
  print(count);
}
```

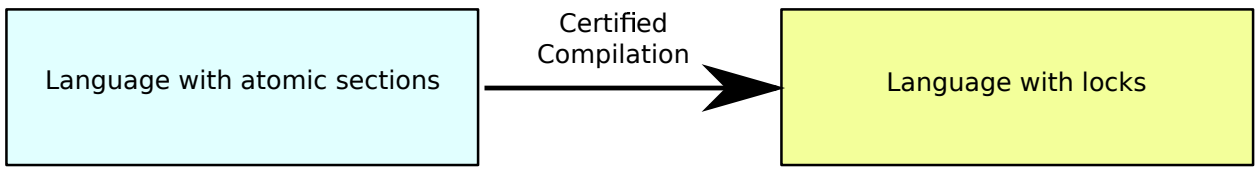
Atomic sections : implementations

- Transaction
  - Inspired from a database managements systems
  - Optimistic approach : assume no interference
  - Cancelled, roll-back and re-execute if interference happens
- Locks inference
  - Pessimistic approach : prevent interference
  - Set of locks inferred by the compiler

We do not focus first on implementation, but on the semantics of atomic sections.

Our goal

Certified compilation of atomic sections toward locks



Nested atomic sections and inner parallelism

- nested for modularity
- distinctive instructions of spawn
- threads live either completely inside or outside a section
- simple, but poor expressivity

Example

```
procedure m1(x)
{
  atomic
  {
    a := [x]
  }
}
```

```
procedure main(){
x := 1;
atomic{
  y := fork (m1,x);
}

}
```

No such constraints, nested and escaping thread. Need to define a new notion of **atomicity**.

Language

- Simple imperative language
- fork/join and atomic primitives
- Nested atomic sections
- Thread can escape surrounding section

Traces

- Abstraction from program semantics.
- Sequence of events.
- events ::= (thread,action)
- Assume disjoint countable set of memory locations, thread identifiers and section names.

Actions

$$
\begin{aligned}
a \quad ::= \quad & \mid \tau \\
& \mid \texttt{alloc}\ \ell\ n \mid \texttt{free}\ \ell \\
& \mid \texttt{read}\ \ell\ n\ v \mid \texttt{write}\ \ell\ n\ v \\
& \mid \texttt{fork}\ t \mid \texttt{join}\ t \\
& \mid \texttt{open}\ p \mid \texttt{close}\ p
\end{aligned}
$$

Well-formed traces

- Set of properties on traces.
- Abstraction of the operational semantics.

Conditions

- Each section name, thread identifier is unique.
- A fork of a thread is done before the action this thread.
- A join on a thread t is done after the action this thread and after its fork.
- The opening and the closing of a section is done by the same thread, and the close matches the last open.

Others conditions, need to have some additional definitions.

$tribe_s$

- for a trace s and a section name, gives the thread identifiers which are allowed to interfere with each other while the section is opened.
- defined as the least set containing :
  - the thread owner of the section
  - the threads forked as a side effect of the execution of the section (tribe children)

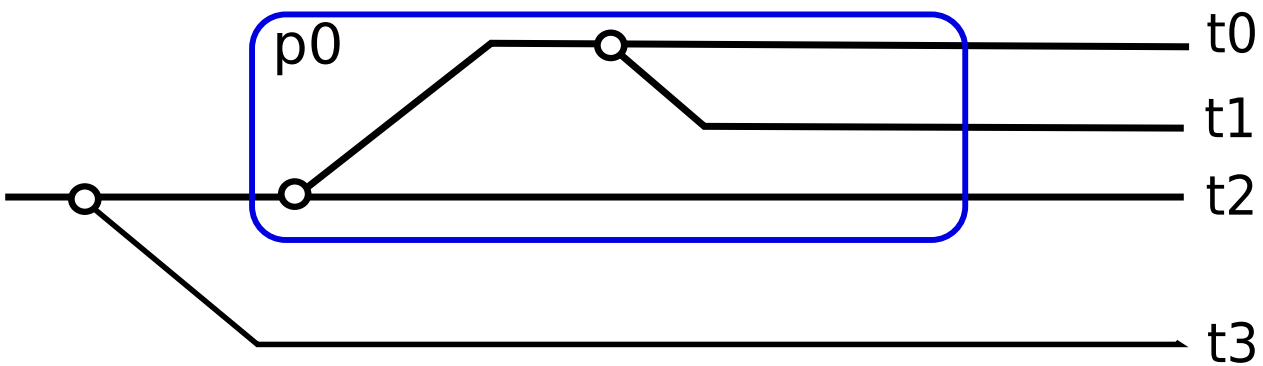


FIGURE : $tribe_s(p_0) = \{t_0, t_1, t_2\}$

subsection

$p' \Subset_s p$

- sections opened by the thread owner of p while p is opened
- sections opened by tribe children of p

concurrent section

$$p \smile_s p' \triangleq p \notin_s p' \wedge p' \notin_s p$$

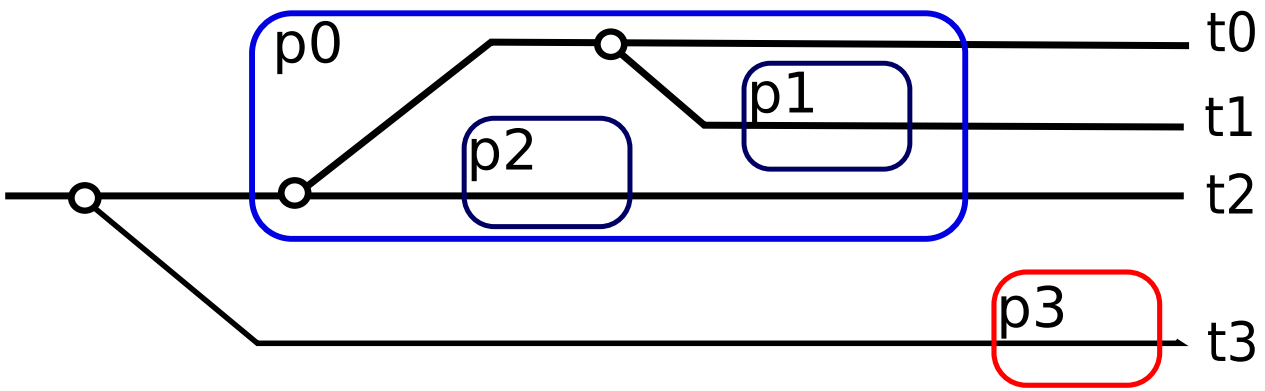


FIGURE : $p_1, p_2 \Subset_s p_0, p_0 \smile_s p_3$

- pending section : a section is pending if there is no closing
- projection : $\pi_s(i)$ i^{th} element of the trace s
- projection : $\pi_s^{act}(i)$ i^{th} action of the trace s

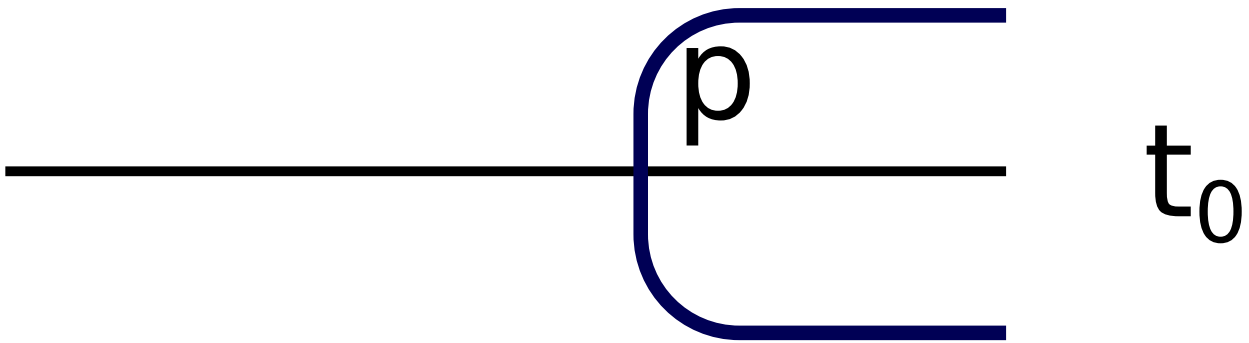


FIGURE : pending section

Well-formed trace(2)

Conditions

- A terminated thread don't have any pending section.
- A thread to join another thread t, must have received explicitly its identifier.
- Two concurrent sections are in mutual exclusion, i.e. the closing of one must precede the opening of the other.

Well-synchronized

We restraint to well-synchronized trace.

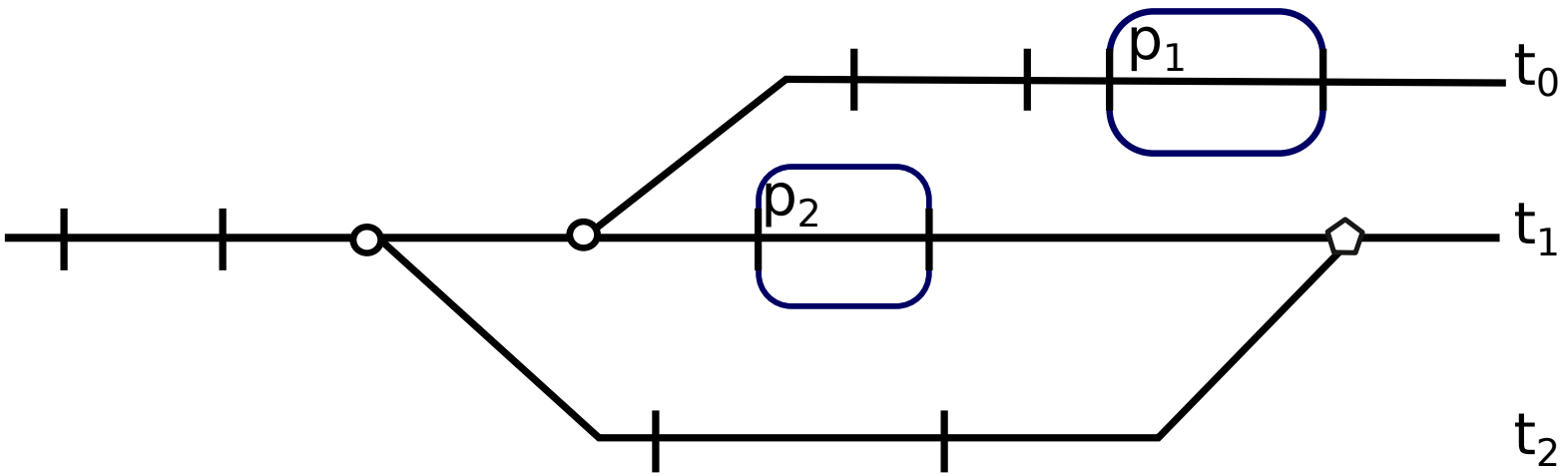
Well-synchronisation

A trace is well-synchronized if all of its conflicting accesses are synchronized

conflicting accesses : two actions conflict if they are both memory accesses over the same location and at least one is a write.

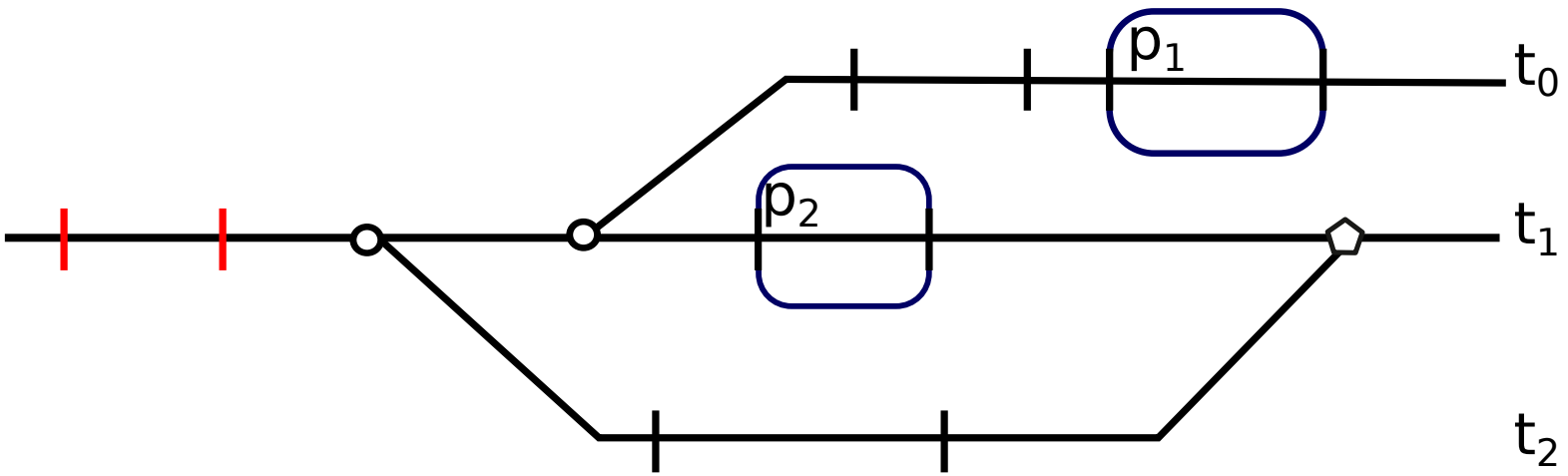
$$\begin{aligned} (\texttt{read}\ \ell\ n\ v) &\bowtie (\texttt{write}\ \ell\ n\ v') \\ (\texttt{write}\ \ell\ n\ v) &\bowtie (\texttt{read}\ \ell\ n\ v') \\ (\texttt{write}\ \ell\ n\ v) &\bowtie (\texttt{write}\ \ell\ n\ v') \end{aligned}$$

Synchronisation



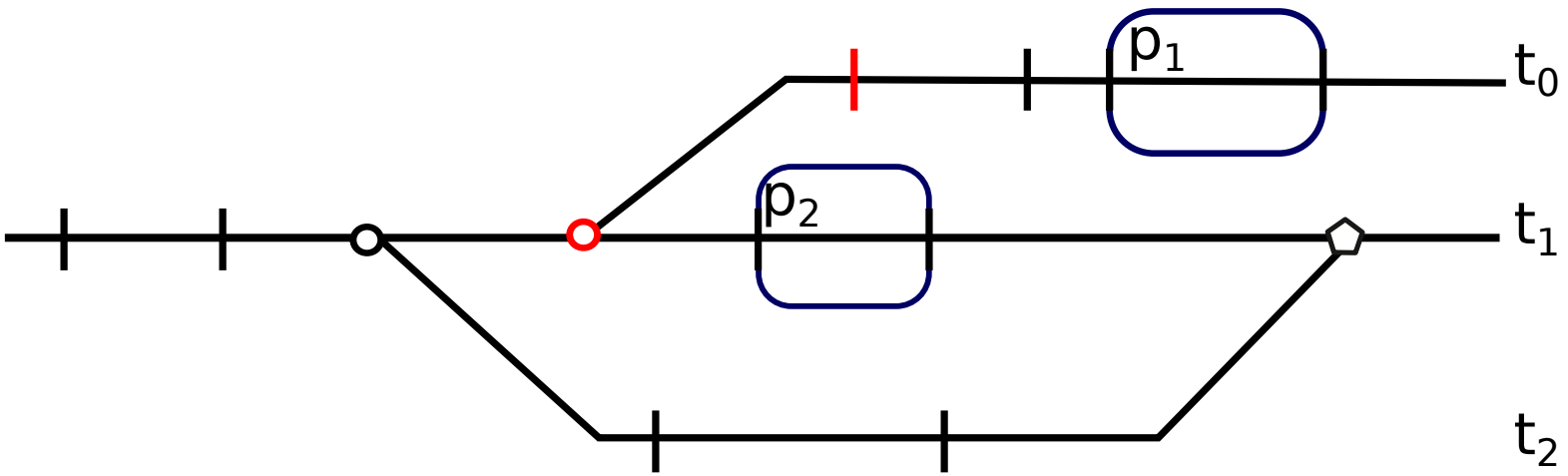
synchronisation

Synchronisation



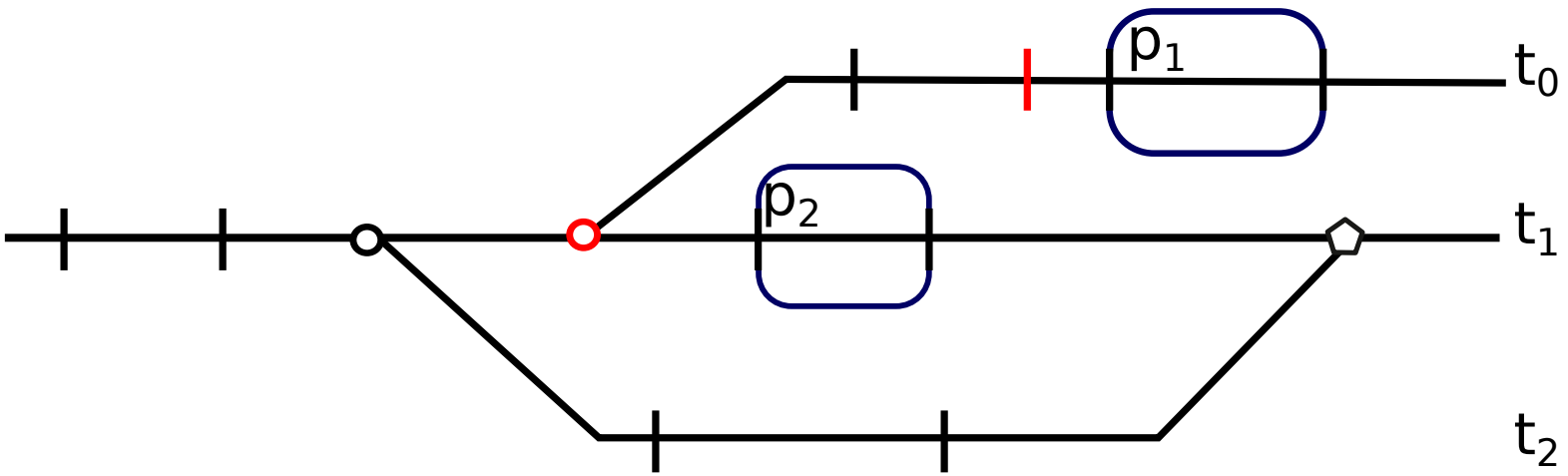
synchronisation of two actions of the same thread

Synchronisation



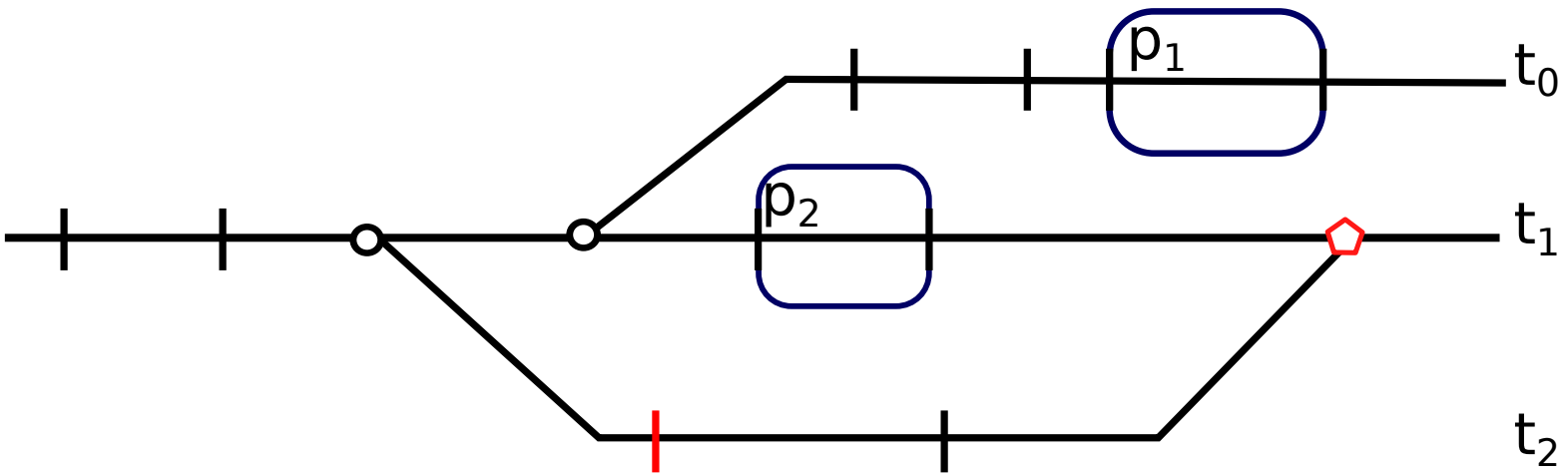
synchronisation of a fork of a thread and the action done by it

Synchronisation



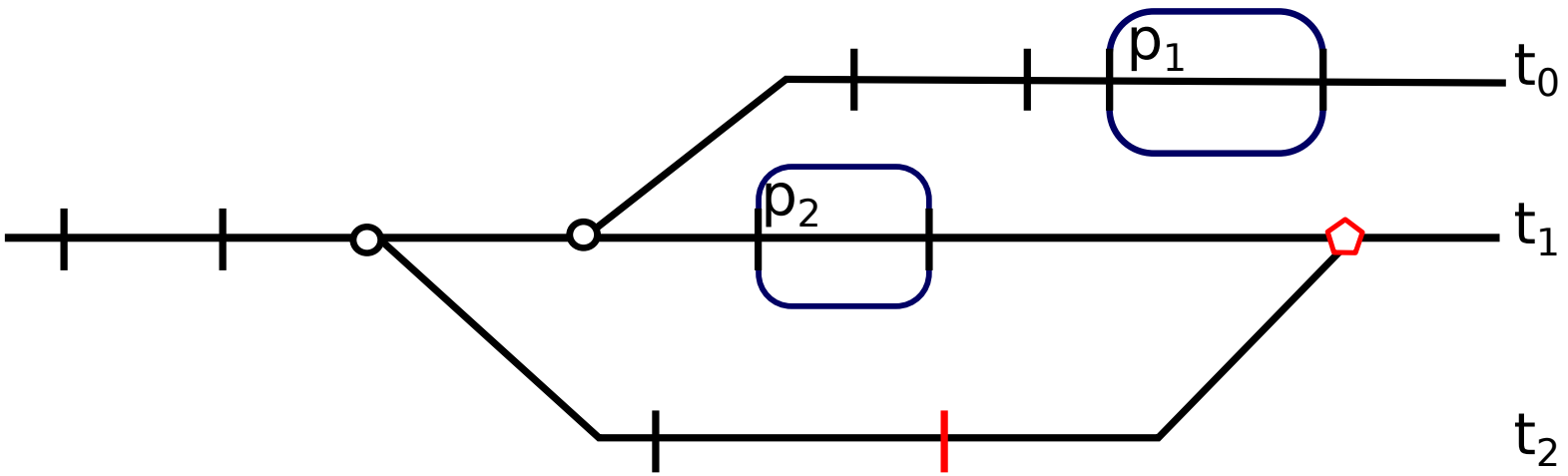
synchronisation of a fork of a thread and the action done by it

Synchronisation



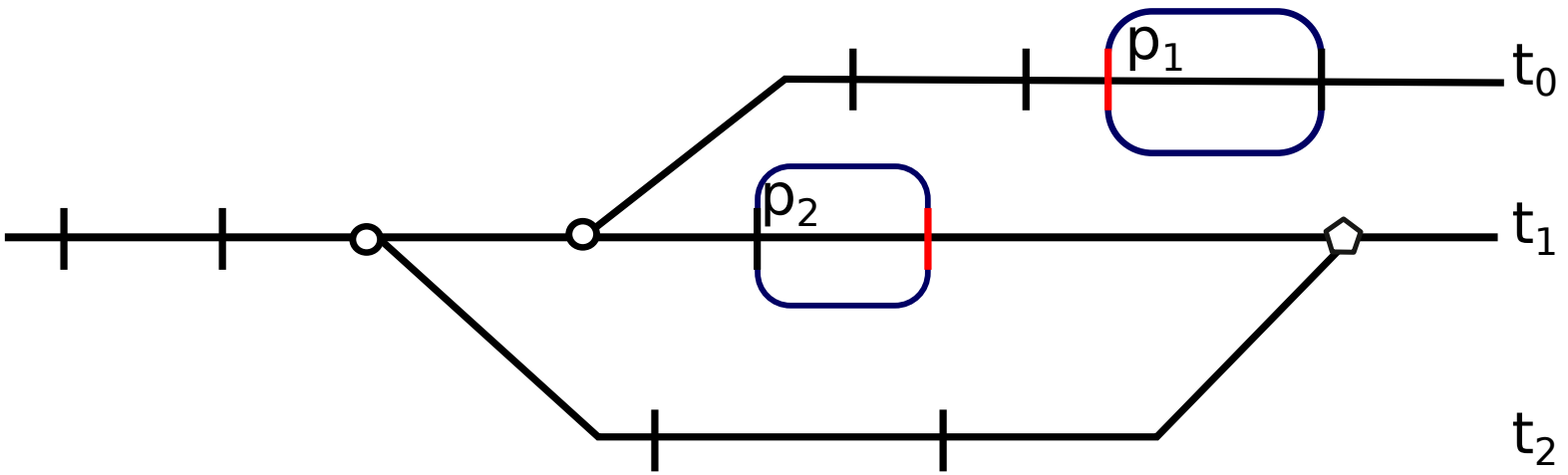
synchronisation of the action done by a thread and the join

Synchronisation



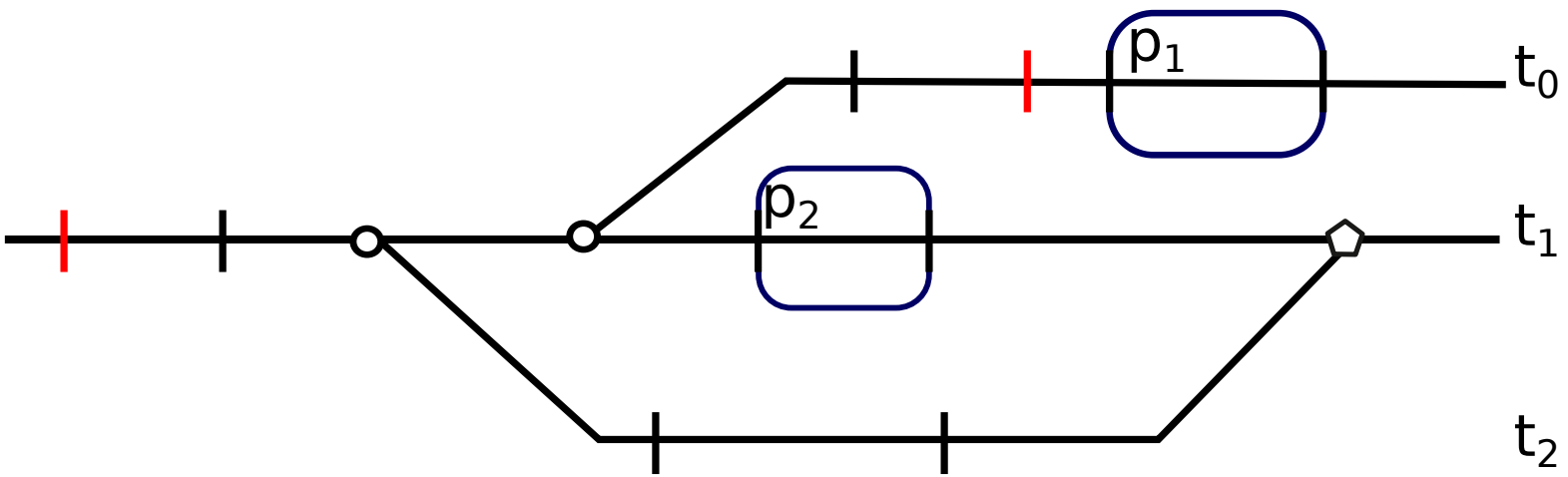
synchronisation of the action done by a thread and the join

Synchronisation



synchronisation of the closing,opening of two concurrent sections

Synchronisation



transitive closure

Atomicity

- Weak atomicity :
  - Atomic section isolated from concurrent atomic sections
  - Ensure by our condition of well-formedness wf
- Strong atomicity :
  - Atomic sections isolated from outside.
  - We want to prove that well-synchronised traces ensure this property of serialisability

Theorem

Every well-formed and well-synchronised trace is serialisable.

Theorem

Every well-formed and well-synchronised trace is **serialisable**.

Serialisable

A trace s is serialisable if there exists a serial trace s' such that s is equivalent to s'.

Serialisable

A trace s is serialisable if there exists a **serial** trace s' such that s is equivalent to s'.

Serial trace

A trace s is *serial* if for all section p only threads members of $tribe_s\ p$ run when the section is opened.

Serialisable

A trace s is serialisable if there exists a **serial** trace s' such that s is **equivalent** to s'.

Serial trace

A trace s is *serial* if for all section p only threads members of $tribe_s\ p$ run when the section is opened.

Equivalence

Two traces s and s' are (schedule-)equivalent [a], noted $s \fallingdotseq s'$, if there exists a bijection γ between positions of s and s' such that

$$\begin{array}{lll} (1) & \pi_s(i) = \pi_{s'}(\gamma(j)) & \text{for all } i < |s| \\ (2) & sw_s\ i\ j \Leftrightarrow sw_{s'}\ \gamma(i)\ \gamma(j) & \text{for all } i, j < |s| \end{array}$$

a. This equivalence relation implies the more classical (conflict-)equivalence relation

Sketch of the proof

Theorem

Every well-formed and well-synchronised trace is serialisable.

Sketch of the proof

- induction on trace s.
  - base case : trivial.
  - $s \cdot (t, a)$ with $s \coloneqq s'$ where s' is serial
    - $s \cdot (t, a) \coloneqq s' \cdot (t, a)$
    - $s' \cdot (t, a)$ serialisable.

Coq

- Formalisation in the proof assistant Coq
- 15 000 lines :
  - 30% definitions
  - 70% proofs

Ongoing work

Properties on traces can be viewed as a specification.

Atomic Fork Join

- Imperative language with atomic sections
- Operational semantics
- Proof that it verifies the specification
- Source language for compilation

Lock Unlock Fork Join

- No atomic sections
- Lock-based implementation
- One lock by level
- Target language for compilation

Results

- Language supporting nesting and escaping thread
- Definition of well-synchronised and serialisability

Perspectives

- Verification of implementation
- Efficient implementation