State Transation Machine for Coq Hackers

Wanted: reactivity

checking time taken to check the whole project

$$\text{checking time} = \text{reaction time} + \text{completion time}$$

reaction time taken by the system to give interactive feedback

completion time taken to check what is missing

Time I want to cut down: reaction time

Incidentally I also cut completion time (quick compilation chain)

Minimizing reaction time

Plan:

check the document out-of-order (relevant to the user first)

Prerequisites (roadmap of this talk):

1. communicate with the UI asynchronously
2. analyze the document (identification of the tasks)
3. model execution of tasks (in the kernel and in OCaml)

1

UI interaction model

Asynchronous feedback

8.4: the PG “protocol” based on REPL

- Synchronous communication
  - `interp : string -> string * id`

8.5: a very conservative “asynchronous” protocol

- Synchronous communication of the document (UI → ITP)
  - `add : string -> id`
  - `edit_at : id -> unit`
- User point of interest (UI → ITP)
  - `goals : id -> goals option`
- Asynchronous feedback (UI ← ITP)
  - `feedback : id -> message -> unit`

Note: by building a protocol on REPL one mixes the declaration of interest and the communication of the document. This “confusion” is problematic: e.g. PG switches off printings for all but the very last command.

2 Formal document analysis

Analysis of the document

The prover must be able to analyse the document to:

- identify the tasks,
- identify dependencies among tasks,
- take scheduling decisions,

before checking it.

What is a task?

The choice depends on:

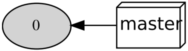
- the language of formal documents:
  which independent parts are clearly delimited
- the runtime:
  which notion of parallelism is offered

We chose:

- task = proof, i.e. the text between `Proof` and `Qed`
- `Qed` is a commitment to not use the proof term

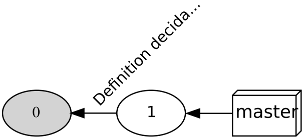
Static analysis (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold  decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



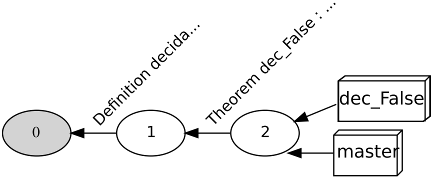
Static analysis (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold  decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



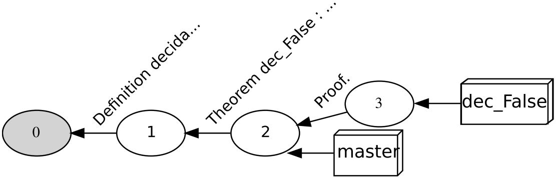
Static analysis (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem  dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold   decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



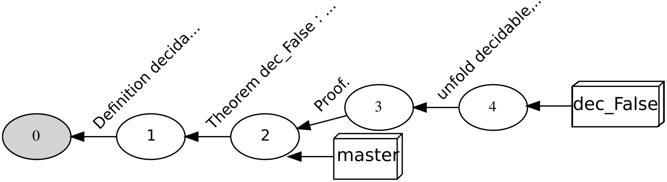
Static analysis (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold  decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



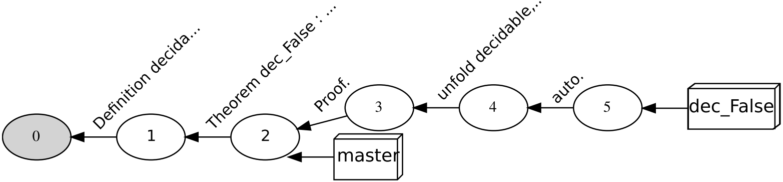
Static analysis (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem  dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold   decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



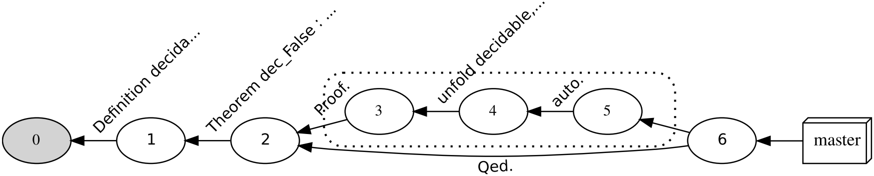
Static analysis (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold  decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



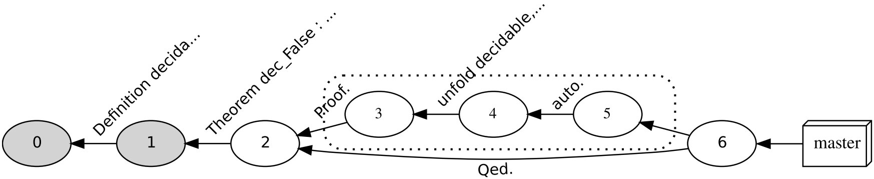
Static analysis (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold  decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



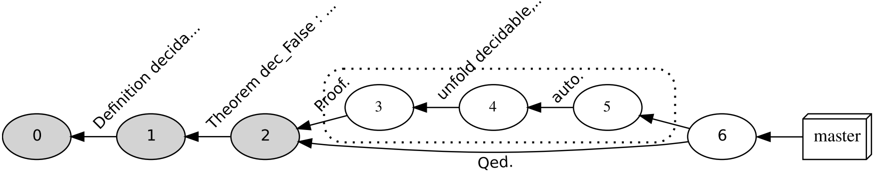
Scheduling (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem  dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold   decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



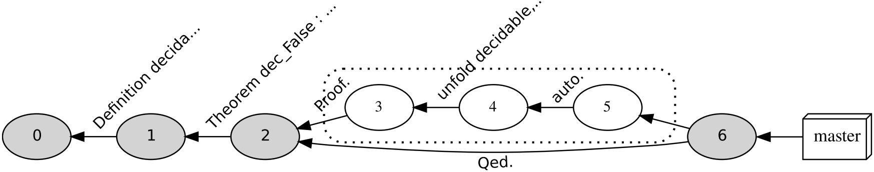
Scheduling (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem  dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold  decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



Scheduling (the State Transaction Machine)

```
(* global *) Definition decidable (P : Prop) := P \/ ~ P.

(* branch *) Theorem dec_False : decidable False.
(* tactic *) Proof.
(* tactic *) unfold  decidable, not.
(* tactic *) auto.
(* merge  *) Qed.
```



3

Asynchronous proofs

Changes to the kernel

Rules in Coq 8.4

$$\frac{E \vdash \mathrm{WF} \quad E \vdash b : T \quad d \notin E}{E \cup (\textbf{definition}\ d : T := b) \vdash \mathrm{WF}}$$

$$\frac{E \vdash \mathrm{WF} \quad E \vdash b : T \quad d \notin E}{E \cup (\textbf{opaque}\ d : T \mid b) \vdash \mathrm{WF}}$$

Rules in Coq 8.5

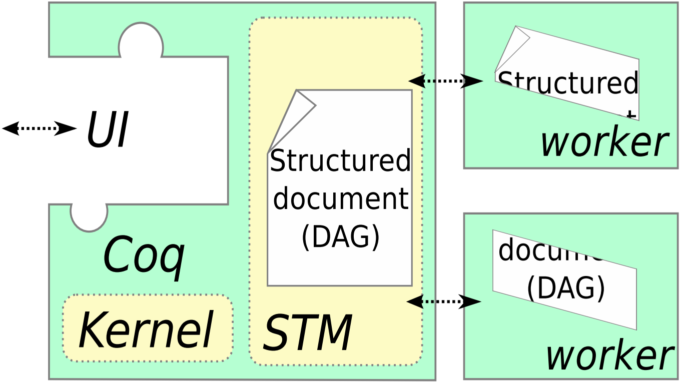
$$\frac{E \vdash \mathrm{AWF} \quad E \vdash b : T \quad d \notin E}{E \cup (\textbf{definition}\ d : T := b) \vdash \mathrm{AWF}} \qquad \frac{E \vdash \mathrm{AWF} \quad d \notin E}{E \cup (\textbf{opaque}\ d : T \mid [f]_E) \vdash \mathrm{AWF}}$$

$$\frac{E \vdash \mathrm{SWF}}{E \cup (\textbf{definition}\ d : T := b) \vdash \mathrm{SWF}} \qquad \frac{E \vdash \mathrm{SWF} \quad b = \mathrm{run}\ f\ \mathrm{in}\ E \quad E \vdash b : T}{E \cup (\textbf{opaque}\ d : T \mid [f]_E) \vdash \mathrm{SWF}}$$

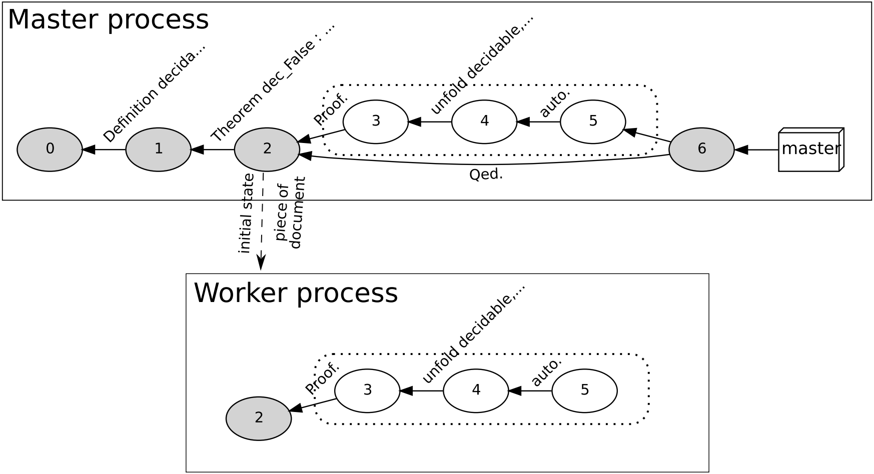
$$\frac{E \vdash \mathrm{AWF} \qquad E \vdash \mathrm{SWF}}{E \vdash \mathrm{WF}}$$

Software Architecture

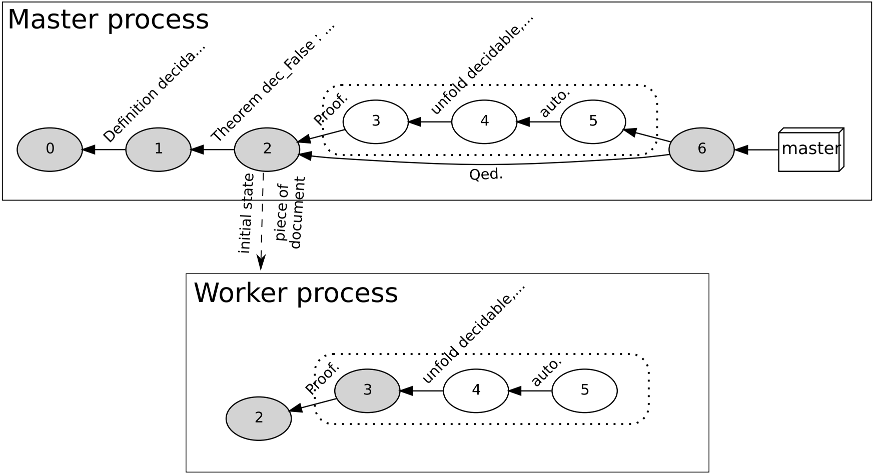
Remember: OCaml has no parallel threads



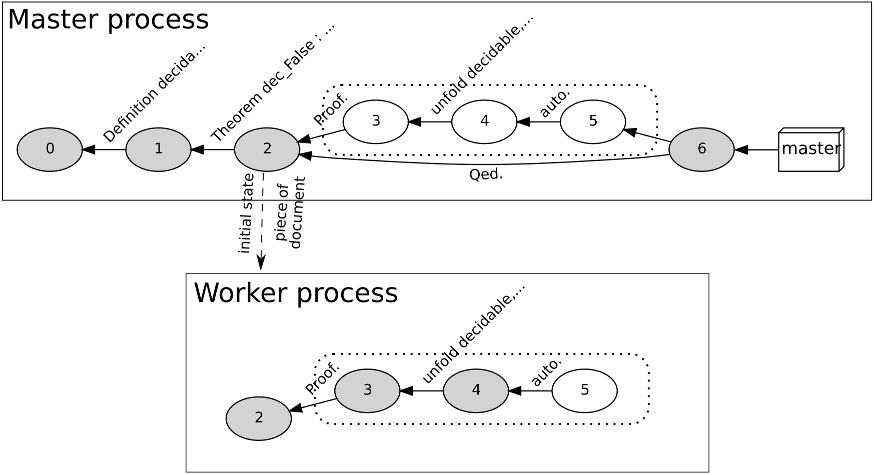
Delegation to a worker



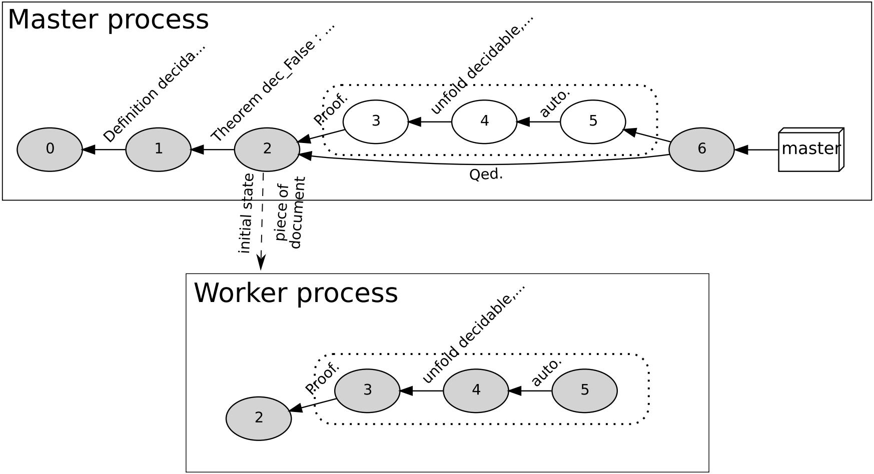
Delegation to a worker



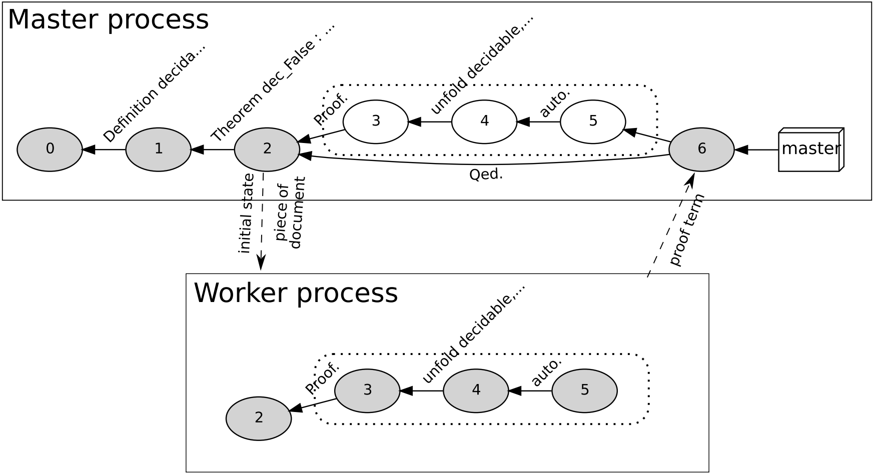
Delegation to a worker



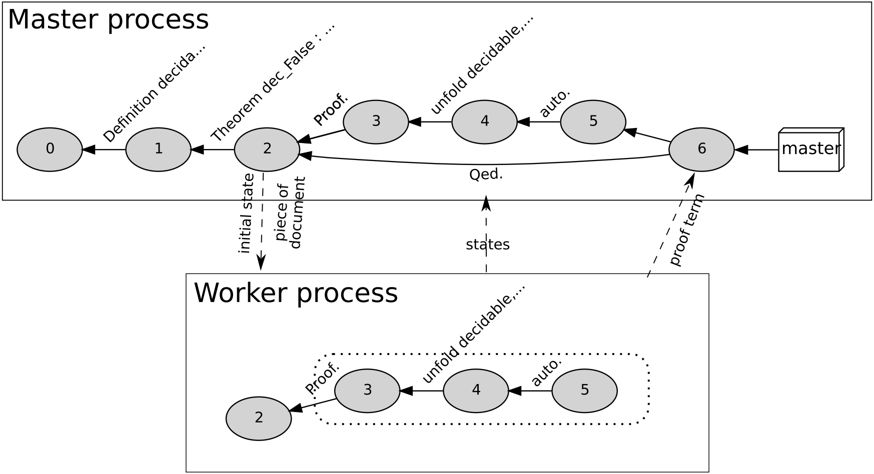
Delegation to a worker



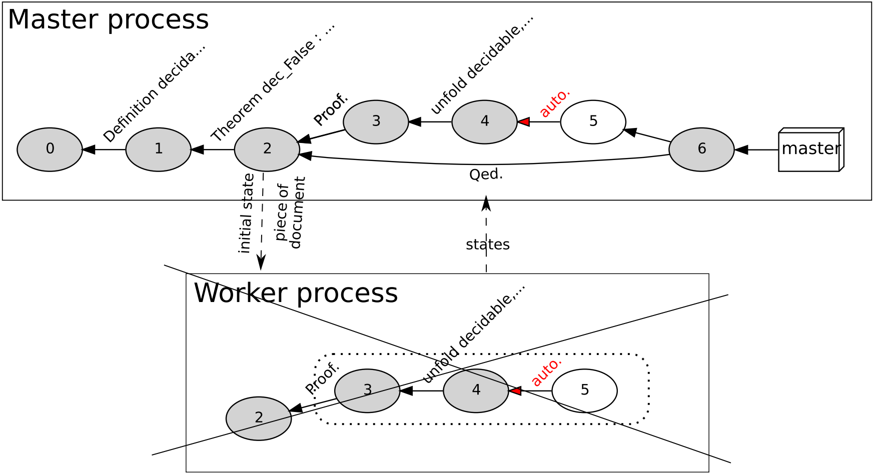
Delegation to a worker



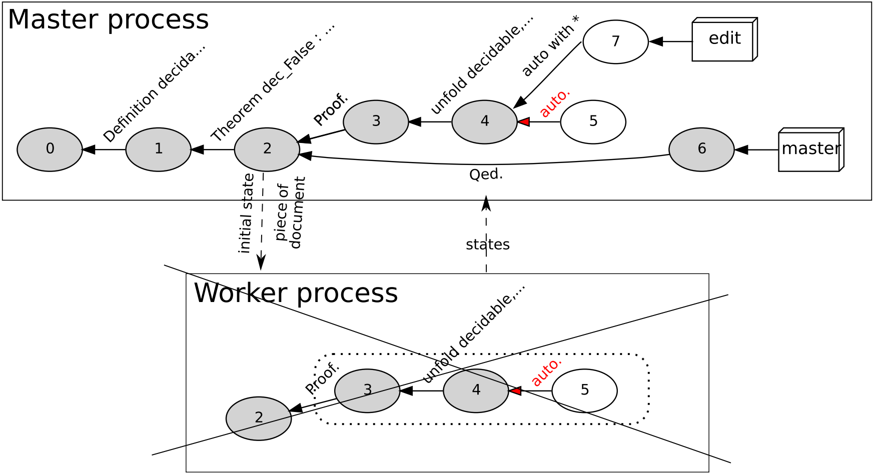
Delegation to a worker



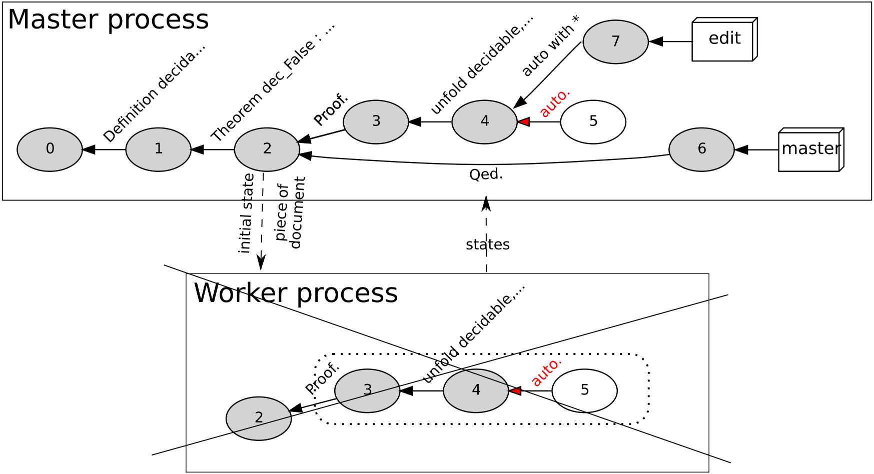
Repairing a proof (in Master)



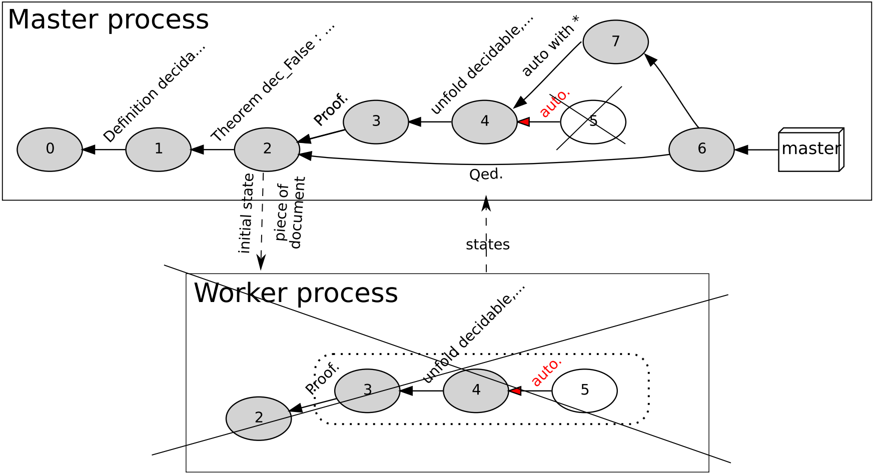
Repairing a proof (in Master)



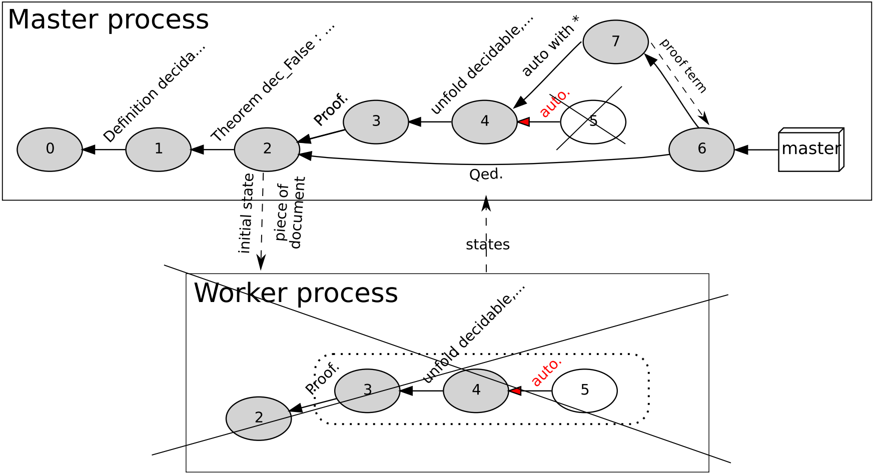
Repairing a proof (in Master)



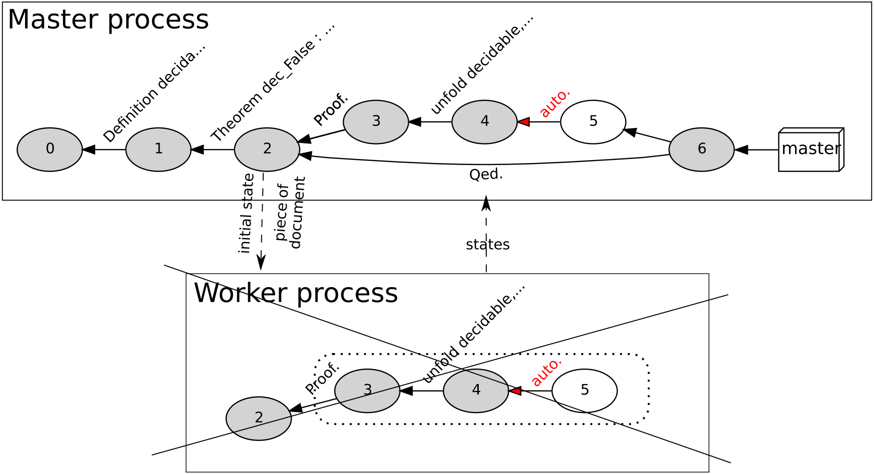
Repairing a proof (in Master)



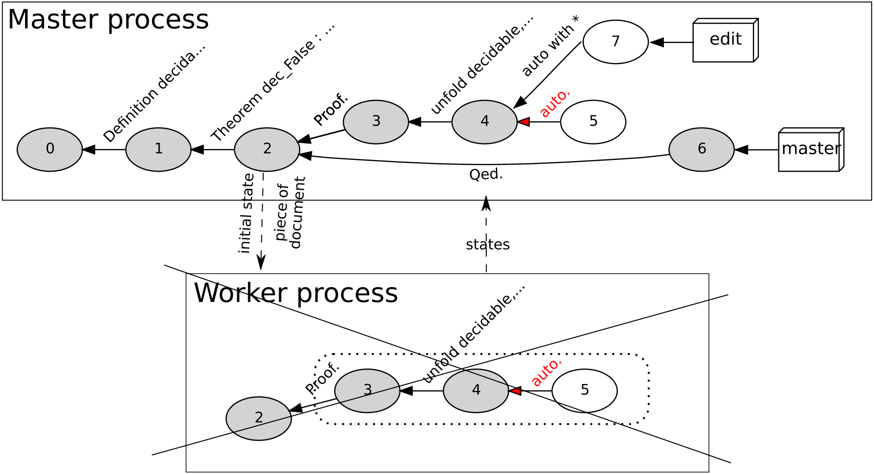
Repairing a proof (in Master)



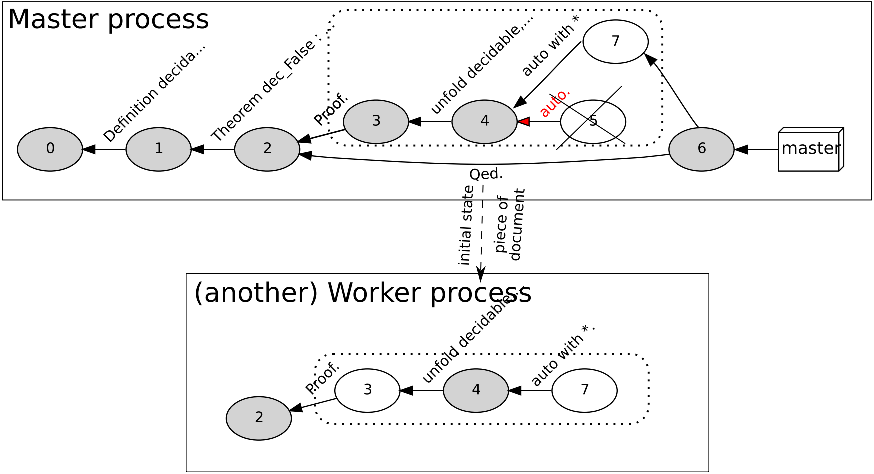
Repairing a proof (in a worker)



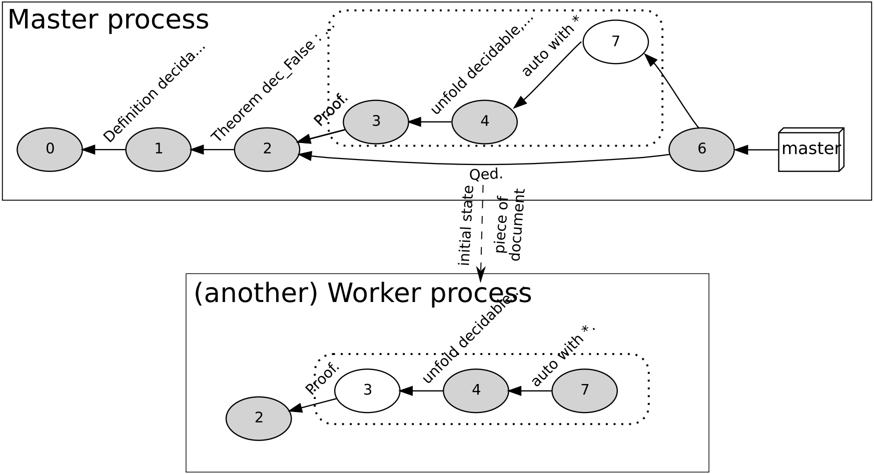
Repairing a proof (in a worker)



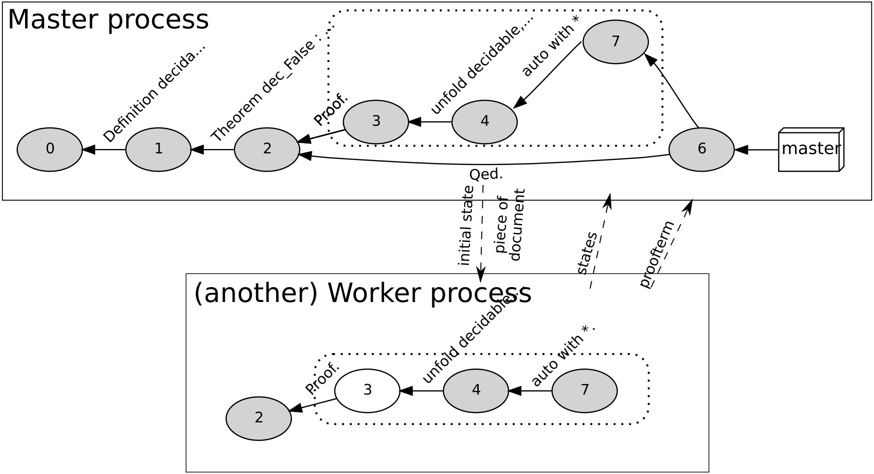
Repairing a proof (in a worker)



Repairing a proof (in a worker)



Repairing a proof (in a worker)



Async task queue API

Used internally to the STM

```
module type Task = sig
 type task
 type request (* marshalable *)
 type response (* marshalable *)
 val request_of_task : [ `Fresh | `Old ] -> task -> request option
 val use_response : task -> response -> [ `Stay | `Reset ]
 val perform : request -> response
end
module MakeQueue(T : Task) : sig (* In the STM *)
 val init : max_workers:int -> unit
 val priority_rel : (T.task -> T.task -> int) -> unit
 val enqueue_task : T.task -> cancel_switch:bool ref -> unit
 val dump : unit -> request list (* -quick *)
end
module MakeWorker(T : Task) : sig (* In the worker *)
 val main_loop : unit -> unit
end
```

The are 3 instances of `Task`: `Proof`, `par:`, and `query` (PIDE only)

4

Next: pull/173

Recovery points

If a sentence fails, do all the following sentences fail?
If a proof step fails, do all the following steps fail?

In 8.5:

- Each task is independent, so failures are local
- Still the whole task is aborted

In pull/173:

- toplevel commands absorb failures occurring before them
- proof blocks confine errors
- demo: `test-suite/interactive/proof_block.v`

Proof Block Detection API I

```
val register_proof_block_delimiter :
  Vernacexpr.proof_block_name ->
  static_block_detection -> dynamic_block_error_recovery -> unit

type static_block_detection =
  document_view -> static_block_declaration option

type document_view = {
  entry_point : document_node;
  prev_node : document_node -> document_node option;
}

type static_block_declaration = {
  start : Stateid.t;
  stop : Stateid.t;
  dynamic_switch : Stateid.t;
  carry_on_data : DynBlockData.t;
}
```

Proof Block Detection API II

```
type recovery_action = {
  base_state : Stateid.t;
  goals_to_admit : Goal.goal list;
  recovery_command : Vernacexpr.vernac_expr option;
}

type dynamic_block_error_recovery =
  static_block_declaration -> [ ‘ValidBlock of recovery_action | ‘Leaks ]
```

End

Thanks for your attention!