

1 Model specification

A model is defined in terms of different roles, where each role corresponds to a generally non deterministic state machine. Non-determinism is a convenient way of acknowledging absents of knowledge due to information only available at runtime.

Events consist the alphabet of the state machines. Events cause state transitions (e.g alter the current state of the state machine). The state after the transition is completed (future state) depends on the state when the event is received (previous state) and the kind of the received event. If there are more than one future states for a transition, the transition is non-deterministic and therefore so is the state machine where the transition occurs.

Beyond altering the state, a transition also invokes one or more operations. An invoked operation in turn creates events or perform an assigned computation. Events are received either from the state machine that dispatched them (self message) or from an other role. Events are synchronous if delivery times preserve the dispatch order (e.g if e_1 is dispatched before e_2 then it is delivered before e_2 as well). Operations can prevent the delivery of dispatched events effectively canceling them.

Transition relations are partially defined if there are state and event pairs for which no transition exists. In this case, the behaviour of the state machine is not predictable. For an asynchronous mode of event delivery, empty transitions are often needed to fully define the transition relation. Empty transitions are those that do not alter the current state nor produce or cancel events.

Transition relations for all roles are specified in a text file with five comma-separated columns:

- Role: The first column defines the state machine in which the specified transition is part of. Identifiers are used to denote the various roles that consist the model.
- Source state: The state where the transition is enabled.
- Event: The trigger of the transition.
- Next state: The target state for the specified transition.
- Operations: A list of operation names that may be accompanied by one or more identifiers enclosed in brackets. The identifiers represent parameter names (e.g. message receiver, reference to a transactional job), that are used as placeholders for code generation. The first operation parameter is implicitly considered to be a unique transaction identifier, which is not written in the specification. In the provided list, operation names are separated by “:”. We use the character “-” for defining transitions with no operations.

A specification excerpt taken from the 2PC implementation for the ACID Sim Tools follows:

```

//when a lock is acquired
_, ST_LOCK , LOCKED          , ST_JOB  , startJob
c, ST_EMPTY, INIT            , ST_INIT , sendInitLog:scheduleTimeout
c, ST_VOTES, OUTCOME_ASKED, ST_VOTES, -
c, ST_VOTES, VOTE_LOGGED   , ST_VOTES, collectVote{id}
w, ST_WAIT , START_JOB     , ST_LOCK , workerLock{job}
w, ST_JOB  , JOB_FINISHED , ST_WAIT , removeJobMessage{msg}

```

Line 1 specifies a comment.

Line 2 defines a state transition that applies to all roles. When a role is in state `ST_LOCK` and receives message `LOCKED`, then the state machine moves to state `ST_JOB` and the transition invokes the operation named `startJob`.

Line 3 defines a state transition from `ST_EMPTY` that for all roles specifies the initial state. The transition regards the "c" role.

The transition of line 4 is a typical example of an empty transition. Transitions that do not have any impact in the execution of the specified model are required to be explicitly defined, in order to ensure that there are no neglected transitions that may be feasible in certain circumstances.

2 Event specification

Except for one, all other events are created by an operation invoked in one or more state transitions. The event that is not produced by a transition initiates the conversation between the roles of the model. Thus, every possible event is a consequence of a past state transition. For model checking purposes, path exploration requires that apart from the state transition relation described in the previous section, the specification needs to provide the events generated or cancelled in each state transition operation. All operations that either create or cancel an event are specified in a text file with four comma-separated columns:

Operation name The name of the operation, without the parameters (if any).

If an operation is invoked in state transitions of different roles, then a separate line is used for each role.

Event The name of an event. If the invoked operation creates an event selected from a set of alternatives, then all possible events are enumerated and are separated by "|". If the operation cancels an event the event name is prefixed by "-" (e.g. -TIMEOUT).

Receiver The role of the event consumer.

Sender The event producer, which may be either a specified role or any other implementation specific component. Implementation components dispatch synchronous events.

A specification excerpt taken from the 2PC implementation for the ACID Sim Tools follows:

```

startJob      , JOB_FINISHED      , c, c
startJob      , JOB_FINISHED      , w, w
sendInitLog   , INIT_LOGGED       , c, lgc
workerLock    , LOCKED            , w, lcw
cancelTimeout , -TIMEOUT          , c, c
nextWorkerJob, FIRST_JOB | START_JOB, w, c

```

Lines 1 and 2 demonstrate a typical case of an operation invoked by state transitions of two different roles. Moreover, we see that in both cases the produced event `JOB_FINISHED` is created and consumed by the same role, therefore represent a self-message. In line 3, as a consequence of the invocation of operation `sendInitLog` by the coordinator role (see line 2 of previous specification excerpt), the "lgc" component replies with event `INIT_LOGGED`. Here, we note that we are only interested in the events consumed by the defined roles and for this reason there is no need to include the event that causes "lgc" to respond with `INIT_LOGGED` (i.e. the coordinator's state transition $(ST_EMPTY, INIT) \rightarrow ST_INIT$). Line 5 shows an operation that cancels an event. When the shown operation is invoked, if `TIMEOUT` has been previously produced, then this event is cancelled and is never consumed. Finally, line 6 shows an operation that non-deterministically produces either the event named `FIRST_JOB` or the event `START_JOB`.

At this point, we have a complete specification of the transaction model of interest, with two sources of non-determinism. The first source is the role specification, where non-determinism is introduced as discussed in the previous section and the second source is the operations with non-deterministic event production, as the one shown in line 6. Path exploration for model checking correctness properties is possible, only if these two kinds of non-determinism are resolved.

3 Input file parsing

In both the role and event specification files, fields are separated by tokens. A utility function, `splitBy`, tokenizes a string based on a character separator.

```

splitBy :: Char → String → [String]
splitBy ch str = case dropWhile (≡ ch) str of
  "" → []
  s1 → w : splitBy ch s2
  where (w, s2) = break (≡ ch) s1

```

While spaces and comments improve readability of the input file, are of no use for code generation and analysis of the state machines.

```

filterComments = filter notComment ◦ map clearSpaces
where
  notComment x = (¬ ◦ null) x ∧ head x ≠ ' '
  clearSpaces = filter (≠ ' ')

```

Methods have a name and a possibly empty list of events produced upon invocation:

```
data Method = Method{
  name :: String,
  events :: [Event]
} deriving (Show)
```

Each input line of the event definition file is converted into a Method structure. All events dispatched by the method are parsed and stored in the events list.

```
parseMethod line = Method (val !! 0)
  (mevents
    (val !! 3)
    (val !! 2)
    (val !! 1)
  )
where
  val = splitBy ', ' line
  mevents s r evs
    | '!' ∉ evs = [Event evs s r ""]
    | otherwise = map (createEv s r) ∘ splitBy ', ' $ evs
  createEv s r n = Event n s r ""
```

Events have a name or a kind that distinguish them. Other attributes are the dispatcher and receiver of the event. Finally, if the first letter of the event is “-”, by convention, the event is considered to cancel the event with the same name. The `ev_gen` attribute of events is used for debugging purposes.

```
data Event = Event{
  ev_name :: String,
  ev_sender :: String,
  ev_receiver :: String,
  ev_gen :: String
} deriving (Show)
```

A transition contains the role where it belongs, a state before the transition took place and one after the transition is completed as well as an event that triggers the transition. Also, since a transition invokes methods that create events, a list contains the events that are produced by the invoked methods.

```
data Transition = Transition{
  role           :: String,
  from_state    :: String,
  to_state      :: String,
  trigger       :: Event,
```

```

pevents          :: [Event]
} deriving (Show)

```

Transition objects are constructed from non-comment lines of input. Since the definition file refers to methods, a list of methods extracted from the event description file is provided. Also, in order to convert the list of methods referred from a transition to a list of events, an other parameter that contains all the known events is provided. This list is derived from the list of methods by the following function:

```

evs = nub ∘ concatMap events

```

The list of events produced by a transition is constructed by joining the lists of events of the invoked methods.

```

method_events meths method_name
= let mname = normalize method_name
  in concatMap (map (λe → e{ ev_gen = mname }) ∘ events)
  ∘ filter ((≡ mname) ∘ name) $ meths
where
  normalize = takeWhile (≠ ' {')

```

However, if a method is invoked by multiple roles, the list will contain events that are not produced by the examined transition. For this reason, the produced list is filtered in order to contain only the events that are valid for the role of the transition.

```

gen_events meths role
= concatMap
  (filter valid_event ∘ method_events meths)
where
  is_cancel = (≡ ' -') ∘ head ∘ ev_name
  valid_event ev = (is_cancel ev ∧ ev_receiver ev ≡ role)
  ∨ ev_sender ev ≡ role
  ∨ (ev_sender ev ≠ role ∧ ev_receiver ev ≡ role)

```

Function fevs, collects all events with a particular name that are received from the examined role. tr_trigger identifies the event that triggers the transition from the list produced by fev. If the event is not produced by a transition, fev will not contain it and a new event is created. The common case is that the event will be found and tr_trigger returns it. tr_events produces the list of events created by the transition.

```

createTransition evs methods st =
  Transition (st !! 0) (st !! 1) (st !! 3)
  tr_trigger tr_events
where
  tr_trigger

```

```

| null fevs = Event (st !! 2) "kkk" (st !! 0) ""
| otherwise = head fevs
fevs = filter (\x → ev_name x ≡ st !! 2
  ∧ ev_receiver x ≡ (st !! 0)) evs
tr_events = gen_events methods (st !! 0)
  ∘ filter (≠ "-") ∘ splitBy ' : ' $ st !! 4

```

The following two functions construct a Method or Transition object respectively for each input line.

```

parse meths = map (createTransition evs' meths ∘ splitBy ' , ' )
  ∘ filterComments ∘ lines
where
  evs' = evs meths
parseMethods = map parseMethod ∘ filter (≠ ∘ null)

```

4 Path exploration

Path exploration is based on calculating the reachability graph for the synchronized product of the role state machines. The defined roles are combined in one state machine where:

- A n-tuple represents the state of the combined state machine. Each element of the tuple is the corresponding state of the individual state machine.
- Events accepted by the individual roles are also accepted by the combined state machine. Transitions changes at most one element of the combined state, namely the element that corresponds to the original receiver of the event.

Roles have a unique name and a current state. The combined state machine is represented by a list where each element is a role.

```

data Role = Role{
  rl_name :: String,
  rl_state :: String
}

```

Scheduled events are stored in a list that from now on is called the future event list (fev). Newly created events are appended to fev. The terminal state of the combined state machine is reached when fev is empty. Events produced by implementation specific components (i.e not roles), are consumed in FIFO order. All possible interlivings of role events are considered for path exploration.

4.1 Optimizations

Examining all possible event interlivings is expensive and for large number of roles or transitions path exploration is rendered impractical. For this reason two optimizations are shown that remove transitions from the roles specification or events from fev as early as possible.

Some cases of non-determinism are irrelevant for path exploration. An example from the 2PC protocol illustrates the case. When the transaction coordinator receives a vote from a participant, it announces the decision to all transaction participants if all participants have voted. Otherwise, the coordinator awaits for the remaining votes. In the second case, the transition caused by the received event produces no events and the state of the coordinator role remains the same. Although the two distinct transitions are important for code generation, only the first case is relevant to path exploration.

An empty transition is defined as:

$$is_empty\ tr = from_state\ tr \equiv to_state\ tr \wedge (null \circ pevents)\ tr$$

From the previous argument, we conclude that a transition is relevant to path exploration only if is not empty, or it is empty and deterministic.

$$remove_emp\ trs = filter\ (\lambda x \rightarrow not_empty\ x \vee deterministic\ x\ trs)\ trs$$

where

$$not_empty = \neg \circ is_empty$$

$$non_deterministic\ x\ y = from_state\ x \equiv from_state\ y$$

$$\wedge trigger\ x \equiv trigger\ y$$

$$\wedge to_state\ x \neq to_state\ y$$

$$deterministic\ x = null \circ filter\ (non_deterministic\ x)$$

Interlivings of events that cause only empty transitions are irrelevant to path exploration and can be ignored. Models with many asynchronous events tend to have quite a few empty transitions and the gain from early removal of irrelevant events from fev is potentially significant. However, an event can be removed only if all possible transitions following a particular state are empty transitions, otherwise a fault is introduced in the model analysis.

The events that trigger an empty transition are:

$$empty_events = nub \circ map\ trigger \circ filter\ is_empty$$

In order to determine the state after which an event can be ignored, we calculate all paths connecting the initial state of the role with the state that consumes the event. Starting from the consuming state, transitions are backwardly followed until the initial state of the role is reached. Transitions triggered by events that cause empty transitions are grouped in two sets, the first contain empty transitions while the second contains normal ones.

$$empty_transitions\ trs = map\ create_starts \circ empty_events\ \$\ trs$$

where

```



```

`tuple_apply` is a utility function that applies a function to both elements of a tuple and returns the result.

```
tuple_apply f (x, y) = (f x, f y)
```

The following function finds for a given role all transitions that move the role state machine to a particular state. Effectively, the function finds all states that precede a given state in all conceivable paths.

```
back_states trs rl st = nub ∘ map from_state ∘ filter (previous_transition st) $ trs
where
  previous_transition st tr = to_state tr ≡ st ∧ role tr ≡ rl

```

Let s_0 be the state that consumes an event and `back_states` find three preceding states, $[s_1, s_2, s_3]$. In order to construct the paths that connect the role's initial state with s_0 , path $[s_0]$ is augmented with the preceding states. As a result, three partial paths are constructed, namely $[s_1s_0]$, $[s_2s_0]$ and $[s_3s_0]$.

```
expand_path trs (e, paths) = (e, tuple_apply expand paths)
where
  augment_path sts = map (flip (:)) sts
  expand = foldr expand_path' []
  back s = filter (flip notElem s) ∘ back_states trs (ev_receiver e)
  expand_path' s@(x : xs) ys
    | (null ∘ back s) x = s : ys
    | otherwise = (augment_path s (back s x)) ++ ys

```

By recursively finding the preceding states of paths and augmenting them, eventually the initial state of the role will be the first state of the path. At that point, the path is complete and can expand no more. When all paths are complete, the function's result is the same as its parameter. Effectively, the list of completed paths is the fix point of the function.

```
fixpoint f x
  | f x ≡ x = x
  | otherwise = fixpoint f (f x)

```

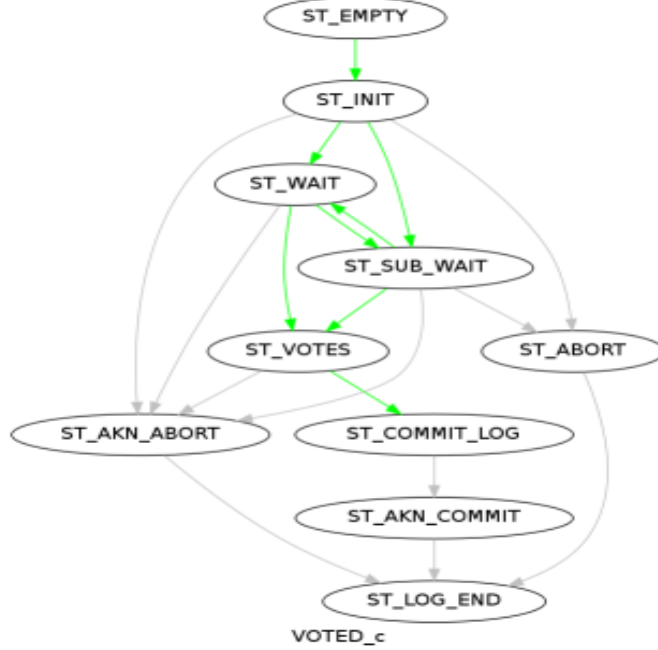
When the complete paths are merged a graph is produced. The following function finds the connected nodes of the graph.

```
merge_paths :: [[String]] → [(String, String)]
merge_paths = foldr union [] ∘ map connect

```

where

$connect\ [x] = [(x, x)]$
 $connect\ l = zip\ l\ \circ\ tail\ \$\ l$



The graph shows all the paths that connect the initial state with the states where event VOTED is consumed. Edges are gray if the transition leads to a state with an empty transition and green otherwise. It is obvious that state ST_AKN_ABORT participates only in empty transition paths while ST_VOTES and ST_WAIT do not. In order to find the states where events can be ignored, two partitions of the graph are found where one contains the wanted states.

We begin by identifying a set of edges (B) from where a green edge initiates.

$starting_points\ (e, (pgray, pblue)) = (e,$
 $filter\ (\lambda x \rightarrow fst\ x \notin\ blue)\ pgray,$
 $map\ fst\ pblue)$

where

$blue = map\ fst\ pblue$

Edges that connect states from B with gray states are removed. In the shown graph, $ST_INIT \rightarrow ST_AKN_ABORT$ and $ST_INIT \rightarrow ST_ABORT$ are removed.

B also includes gray states that initiate an edge which ends to a state in B ($s_0 \rightarrow s_B$).

$nempty\ (e, gray, blue) =$
let

```

    not_empty = filter (λx → snd x ∈ blue) gray
    blue_states = map fst not_empty
    blue' = union blue blue_states
    gray' = filter (flip notElem not_empty) gray
  in (e, gray', blue')

```

In the presented example, nempty would not add any edges, since there are no edges that initiate from gray states and end to green states. In general, the fix point of nempty results to the two wanted partitions of the graph for a given event. Function empty_states finds the fix point of nempty and returns the states where an event may be ignored.

```

empty_states ev =
  let (e, gray, blue) = fixpoint nempty (starting_points ◦ merge $ ev)
      all_gray = unzip gray
  in (e, nub $ fst all_gray `union` snd all_gray)
  where
    merge (e, paths) = (e, tuple_apply merge_paths paths)

```

4.2 Path exploration

Each role in the combined state machine has a state that is represented by the following data structure.

```

data State = State{
  ns_name :: String,
  ns_role :: String
} deriving (Eq)
data PathElement = Tr State State Event deriving (Eq)

```

A series of PathElement constitute a path. Paths that connect the initial state of the combined state machine with the terminal state, are complete. Only complete paths are considered for property verification.

```

type TrPath = [PathElement]

```

Depth first traversal of the state transitions of the combined state machine is employed for path exploration. Information available for the combined state machine include:

- Transitions of all individual role state machines.
- The combined roles. Implementation specific components are not included in the roles list.
- A list of states where an event may be ignored
- Scheduled future events which we refer as fev.

- A list of traversed states. When a new state is visited, it is added to this path. States that are already visited are not visited again to prevent circles and endless recursion. After all events of fev are consumed, the path is complete.
- A function which decides to ignore an event based on the previously visited states stored in path.
- An event that triggers a transition for this path.
- A list of calculated completed paths. Complete paths are appended to this list.

Transitions of the combined state machine regard the role that receives the triggering event. No other roles are directly affected by the event. From now on, we refer as “current’ the event receiving role and it’s properties (e.g current state).