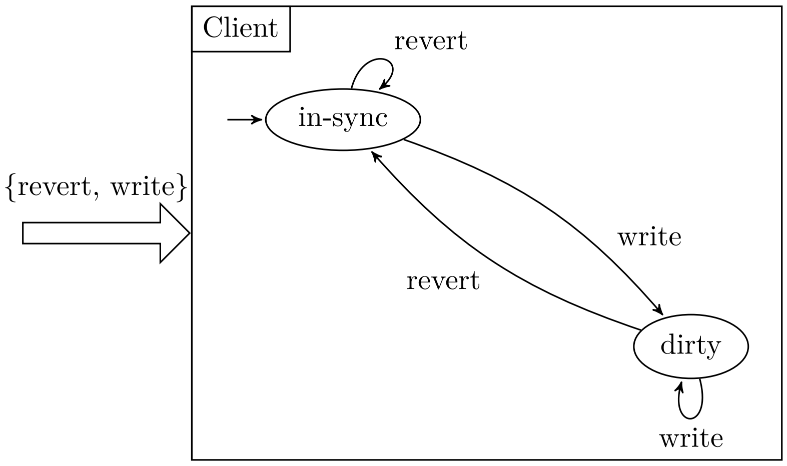
3rd Seminar – Process models, cooperating behaviour models – Solutions

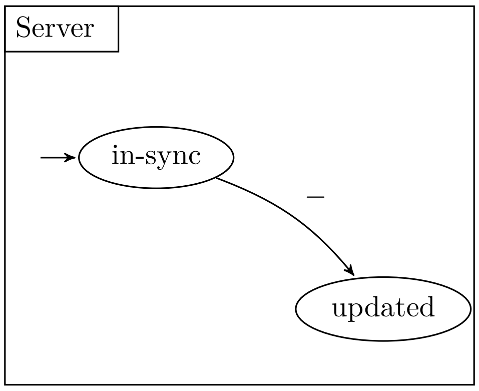
1 Modelling a complex system

We are modelling a cloud-based data storage (e.g. Dropbox, Google Drive, Tresorit) with only one file. Both the server and the client (e.g. a laptop) has a replica of the file, initially with *identical* contents. Modifications of the file are transmitted via *synchronization*. A *conflict* occurs if both instances are modified before synchronization and the user has to resolve this conflict on the client.
The file can become modified locally on the client or on the server (e.g. due to the work of another client). The client and server may synchronize their content due to a user command or spontaneously from time to time. At that point the changes (if there is any) are transmitted to the other file and the replicas became *identical* again. However, if both files have been modified independently since the last synchronization, then a *conflict* occurs. In this case the client compares the local and remote versions of the file and the user must resolve the conflict.

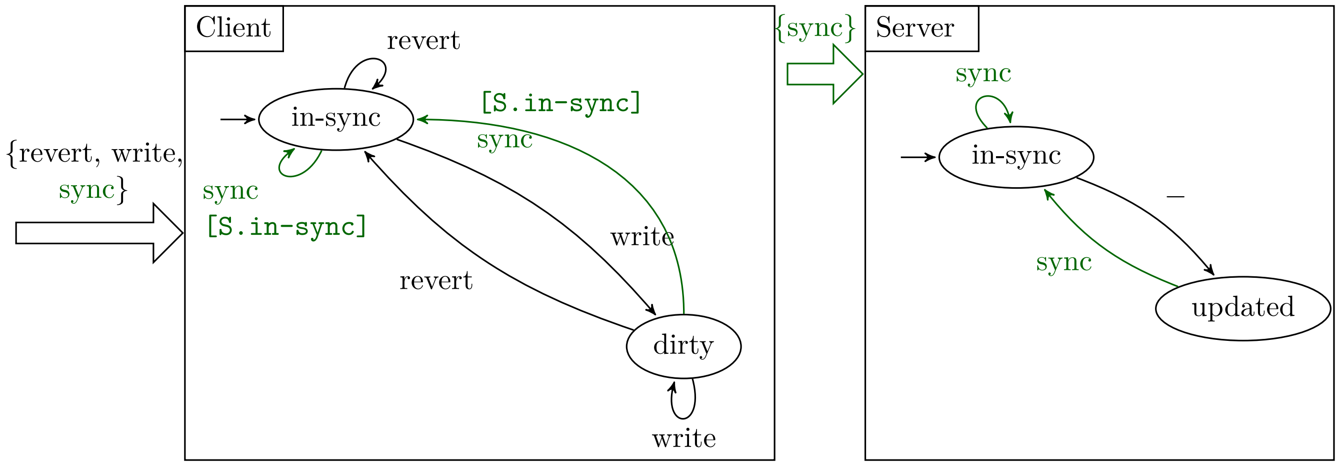
a. Model the (partial) behaviour of the laptop client with a state machine. Initially the client is in *identical* state (the local file is identical to the one on the server at the time of the last synchronization), but the *write* input causes it to transition to *dirty* state (and it stays there after further *write* inputs). As a result of a *revert* input, it moves from any state to *identical* state.
Solution
Custom notations:
   - identical state: *in-sync*
   - synchronize event: *sync*



b. The possible states of the server (regarding only the synchronization with the client) include *identical* and *updated*. A different user (or the same user using another client, e.g., a smart phone) with write permissions might update the replica on the server. At the beginning, the server is in state *identical*.
Solution
Here the interesting part is the spontaneous transition (with an empty trigger/input event). This is due to our decision to model only one client. This empty input event is essentially the sync event of other clients.

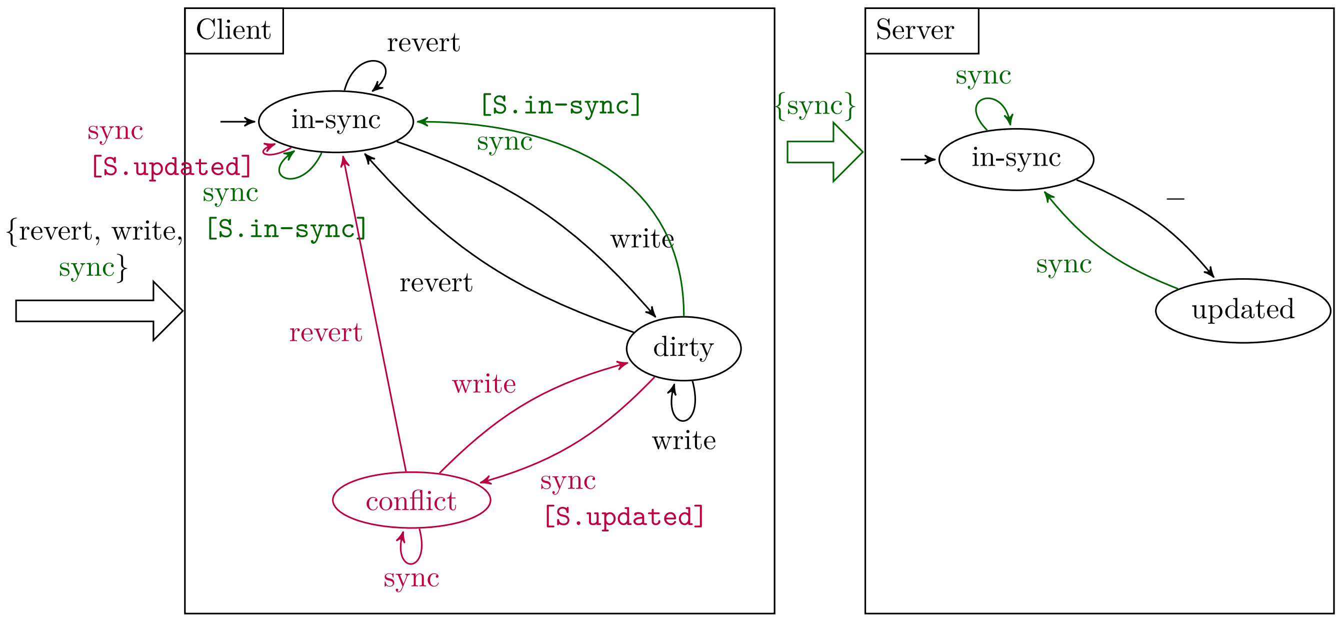


c. If the server is in *identical* state, then as a result of *synchronize* input the client uploads the local modifications (if there is any) to the server and moves to *identical* state. The server also receives the *synchronize* input. Where do the two automata cooperate?
Solution
We put the guard condition `[S.in-sync]` on the two new transitions of the Client. Interesting to note is that this depends on the actual state of the other state machine – so this is a *cooperation*. This is an abstraction of reality, since in real life we need messages between the client and server to determine the other's state (or to exchange file versions).

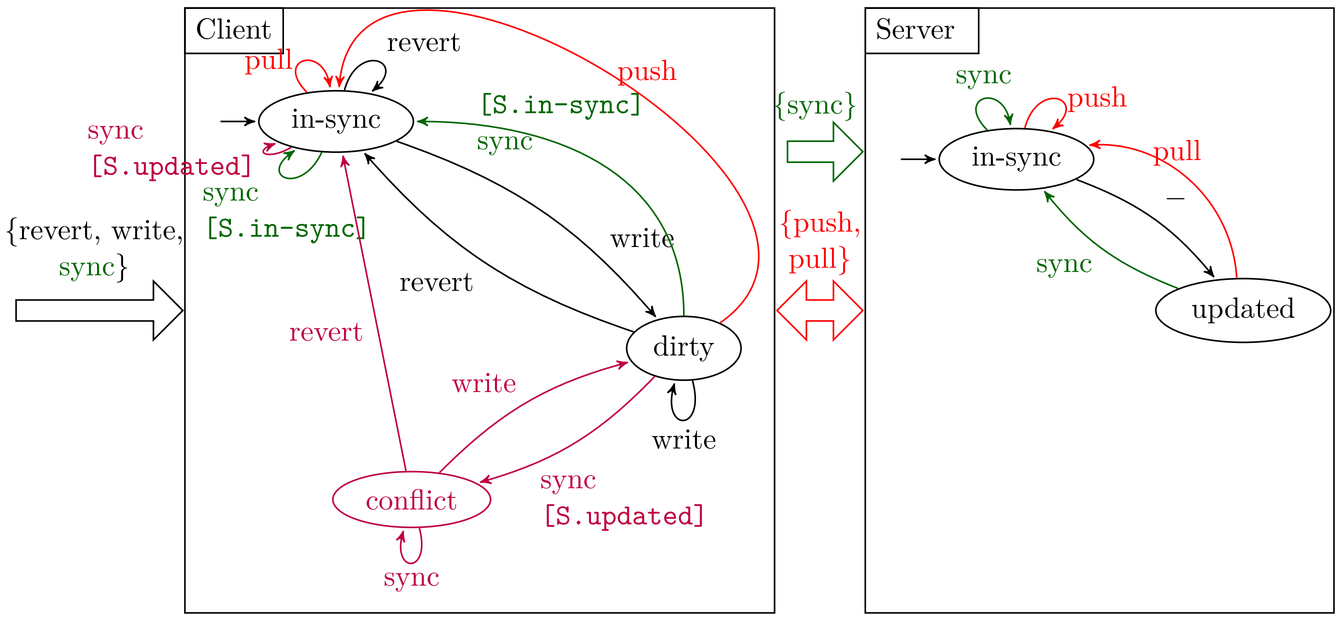


d. If the server's current state is *updated*, then, as a result of *synchronize* input given on the client, the server transitions to *identical* state. The client either stays in *identical* state, or transitions from *dirty* to *conflict* state. What does this mean? What should happen in the conflict state? Where do the two state machines cooperate?
Solution
This means that the two automata step together, that is the *sync* input event affects both automata!
(Note: in this case we say that the product automaton is the *mixed product* of the two individual automata. They step mainly asynchronously, but sometimes they perform synchronized steps, thus the name mixed product.)
We put the guard condition `[S.updated]` on the two new transitions of the Client. Notice that there are two transition originating from the *Client.in-sync* state and their guard conditions are the *complements* of each other (since the Server has only two states): `[S.in-sync]` and `[S.updated]`. Accordingly these transition can be merged into a single transition, one without a guard condition (this isn't denoted on the figure).
It's the Client's responsibility to resolve the conflict when it detects one. The Server goes back to *in-sync* state since it hasn't received a newer version since the last synchronization. Possible ways to resolve the *conflict* state:

- revert $\rightarrow$ in-sync
- write $\rightarrow$ dirty
- sync $\rightarrow$ conflict

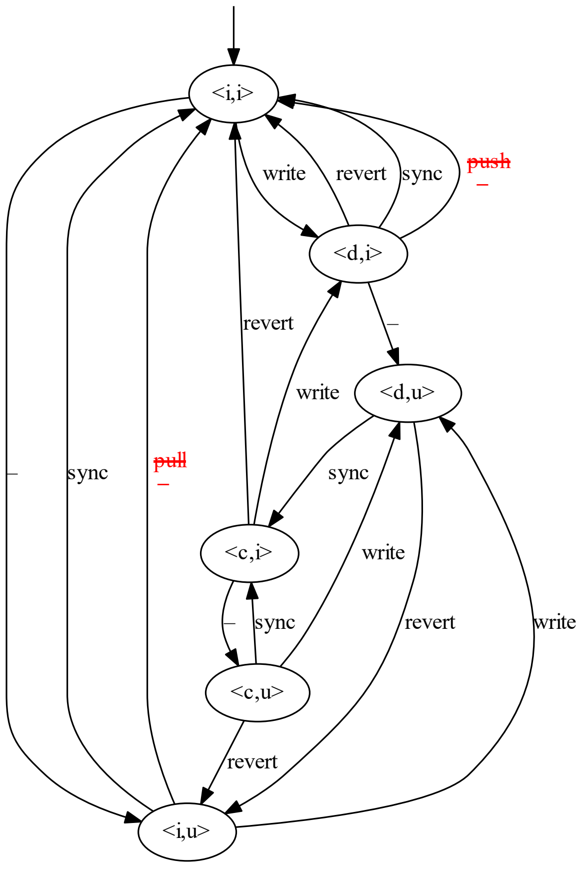


e. Sometimes the client synchronizes with the server by itself, without any user input. What does it mean? Where do the two state machines cooperate?
Solution
It's the same as before, but instead of an outer event (*sync*) we have *inner* events (lets call these pull and push). Optional modification: ensure that automatic synchronization never causes conflict.



f. Create the product state space of the client-server system based on the two automata in the mixed product.
Solution
The states of the composed state machine are described by a vector of two elements ($\langle \text{Client}, \text{Cerver} \rangle$). Beginning with the initial state ($\langle \text{in-sync}, \text{in-sync} \rangle$), we can record in a table for for each possible events, what will be the next state. We add a new row to the table for each new "next states". In case of a rendezvous event the composed state machine will only make a step, if both original automata could make a step for that event in their corresponding states. (In the table, we have showed loop transitions with an $*$.)

	asynchronous (private) events		common (shared) events	rendezvous events		spontaneous transitions
	revert	write	sync	push	pull	–
⟨in-sync, in-sync⟩	$*$	⟨d, i⟩	$*$	—	—	⟨i, u⟩
⟨dirty, in-sync⟩	⟨i, i⟩	$*$	⟨i, i⟩	⟨i, i⟩	—	⟨d, u⟩
⟨in-sync, updated⟩	$*$	⟨d, u⟩	⟨i, i⟩	—	⟨i, i⟩	$*$
⟨dirty, updated⟩	⟨i, u⟩	$*$	⟨c, i⟩	—	—	$*$
⟨conflict, in-sync⟩	⟨i, i⟩	⟨d, i⟩	$*$	—	—	⟨c, u⟩
⟨conflict, updated⟩	⟨i, u⟩	⟨d, u⟩	⟨c, i⟩	—	—	$*$



Please notice that the red transitions in the figure are shown to be spontaneous, instead of having a rendezvous trigger event. That is, because the rendezvous events represent some inner events, and they have names only two show the cohesive spontaneous transitions in the different state machines. In the single composed state machine we do not need these names to describe the synchronized spontaneous transition. (For the sake of clarity loop transitions are not shown on the figure.)

g. (*Bonus task*) The server and client can check each other's state directly in this model and the synchronization also happens instantaneously. However, the communication between the client and the server is implemented with messages in a real life distributed system. Some time elapses between sending the message and receiving an answer. How can we refine the model in order to incorporate this behaviour?

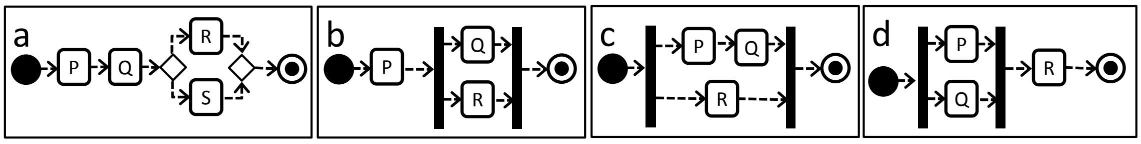
Solution

Homework.

2 Process execution

We observed every step during the execution of a process. We detected the following event sequence:

> Process started, *P* started, *P* completed, *Q* started, *R* started, *Q* completed, *R* completed, Process completed.



Which ones can be valid models of the observed system from the business process models a, b, c, d?
Solution
Assume that every activity takes some time to execute (i.e. they are not atomic). The business process models b and c are valid models of the system. In case of models a and d the completion of activity Q must precede the start of activity R. This constraint is violated by the observed event sequence.

3 Control flow based on source code

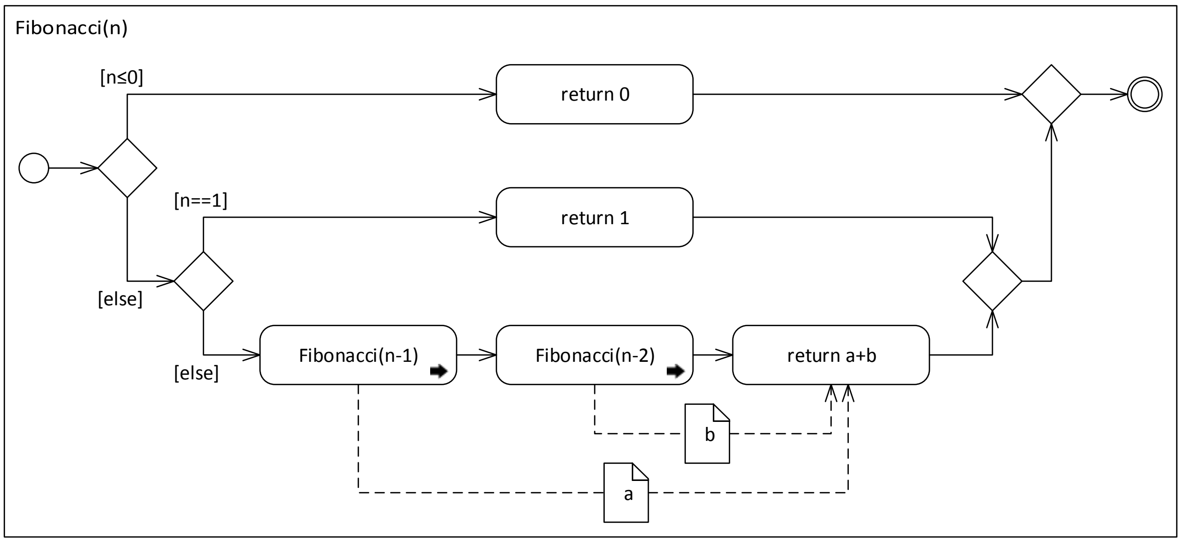
Consider the following function written in the C programming language.

```c
unsigned long long fibonacci(int n)
{
   if (n <= 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   } else {
      unsigned long long a = f(n - 1);
      unsigned long long b = f(n - 2);
      return a + b;
   }
}
```

a. What is the control flow defined by the function?

Solution

We denote the recursive call with a "call" element (there is an arrow in the corner of the box).



b. Check whether the process is well-structured or not!

Solution

We check whether every block is well-structured starting from inside (with the individual activities) and heading outwards (towards the whole process):

- (An empty activity is a well-structured block.)
- An activity with one input and one output is a well-structured block, thus every activity in the process is well-structured. (This can be stated conditionally only, depending on whether the referenced processes are well-structured.)
- The sequence of three well-structured blocks (single activities) is a well-structured block.
- Between the inner decision node (with the conditions `n == 1` and `else`) and its merge node pair every path contains a well-structured block, thus the inner decision-merge node pair as a block is also well-structured.
- Between the outer decision node (with the conditions `n <= 0` and `else`) and its merge node pair every path contains a well-structured block, thus the outer decision-merge node pair as a block is also well-structured.
- Between the *only* start and the *only* end node there is a well-structured block, thus *the whole process is well-structured.*

c. Identify the data dependencies (data flow) between the activities!

Solution

There are data flows (denoted as dashed arrows in the figure) from the two recursive call to the returned sum of previous results. The point is that the result of the first recursive call is not needed to make the second recursive call.

d. If the programming language or the runtime environment allows it, where can we parallelize the program?

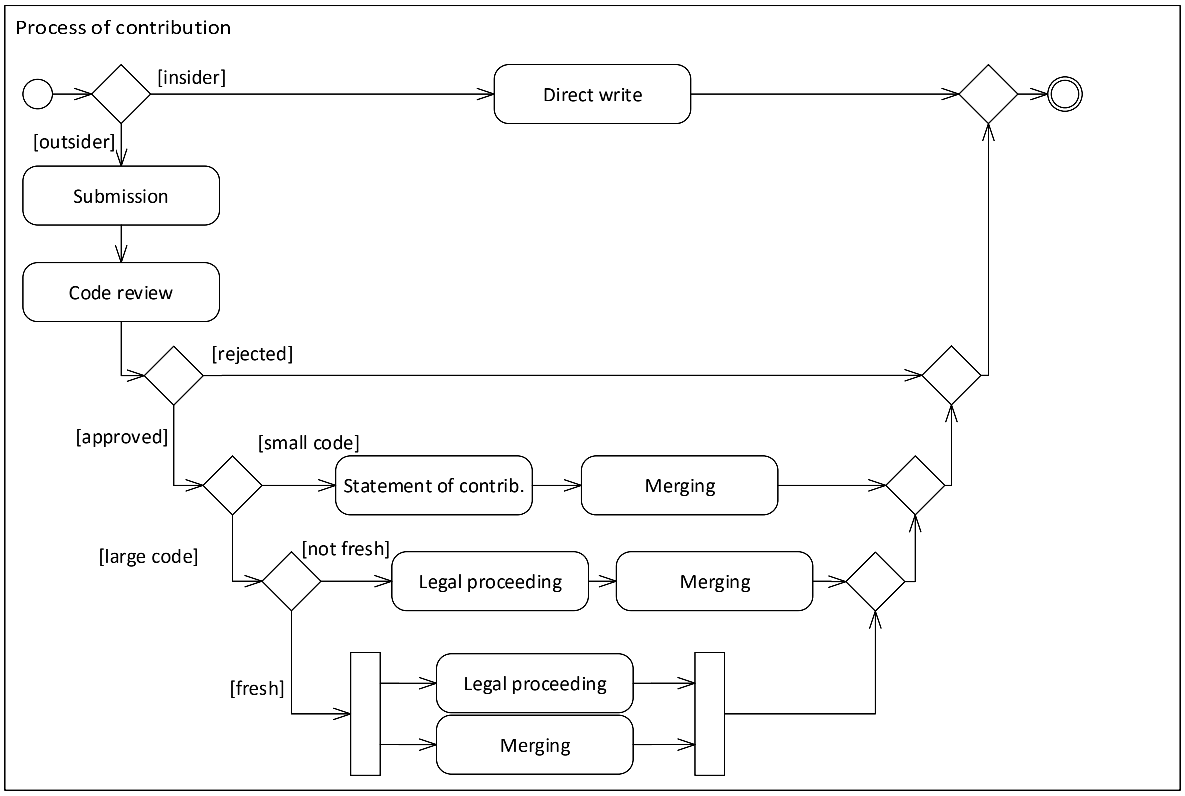
Solution
Based on the solution in task c., the location of parallelization is trivial: the two recursive calls can be made in parallel.

e. (*Bonus task*) What ensures the termination of the function?
Solution
The value of variable `n` is decremented with every recursive call. Accordingly, the condition `n <= 0` (or `n == 1`) will hold sooner or later, thus the process will take the return path.

4 Control flow based on textual specification

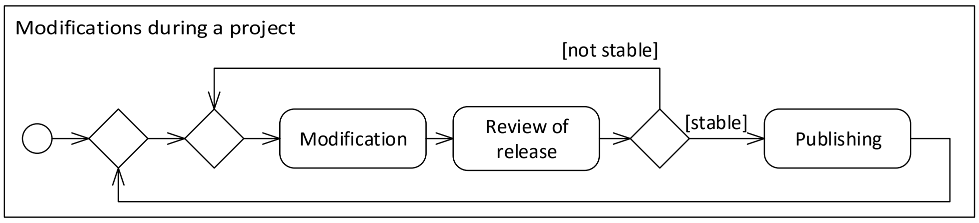
The code repository (e.g. Git) of a big software foundation is home to the development of numerous open-source software. In addition to the reliable internal developers, outsiders can also send bugfixes or newly implemented features. It must be ensured that the published software only contains legitimately (e.g. with the consent of the employer) contributed source code.

a. If a developer would like to contribute to a project he or she must take some steps based on his or her status. Internal developers can write directly to the section of the code repository reserved for the specific project. Outsiders first have to submit their code to *code review*, after which an internal developer has to inspect it, and then either reject or approve it. If the code contributed by the outsider is sufficiently short (e.g. a small bugfix), then he or she only has to make a short statement of contribution in order to merge his or her code into the repository.
In case of large outsider contributions (e.g. integrating a completely new module) the merging process is different. After the insider approval, the legal department of the foundation clarifies the legal status of intellectual property of the modifications via a dedicated administrative proceeding. Only after the successful closure of this process may the internal developer merge the code. Here, they make an exception in the case of freshly started projects, still before their first official release: merging the approved code into the repository does not need to wait until this legal proceeding is done. Create a process model based on these activities and constraints.
Solution



b. The development of a software project involves repeatedly modifying the source code, until the project management decides that the software is stable enough for an official release. At this point they publish a new stable version of the software, then it's the developers' turn again, and so on. Create a process model based on these activities.

Solution



By *Modification* we mean the action when developers create/modify code and submit it.
Notice that the model focuses only on the life cycle of the project and doesn't contain the participants/actors (denoted in the textual specification).

c. (*Bonus task*) Check whether the processes are well-structured or not!
Solution
Use the same technique as in exercise 3. The first process is well-structured. The second is *not*, since there is *no end node*. (The activities and the loops of the second process build a well-structured block, but the end node of the process is missing to make the whole process well-structured.)

d. (*Bonus task*) What is the relationship between the process models designed in the previous sub-tasks?
Solution
They show the life path of two different things in the same system. There can be weird overlaps in the modelled processes, e.g., code review lasting through a release cycle. But there is no direct relation between the two models.