SOEN331: Introduction to Formal Methods for Software Engineering
Assignment 3: Extended Finite State Machines

Prepared by: S. Symeonidis
Instructor: C. Constantinides

March 10, 2015

1 Introduction

A deadly virus of unknown origin has plagued the world and has infected 98% of the population. The remaining 2% of humanity would not be categorized as living — they were desperately surviving.

Casual chatter, and the cacophony of traffic in heavily urbanized areas have been replaced by the howling winds, rushing through the ruins, which are now the only proof that humanity once was.

Hordes of bandits, and smaller groups of scavangers try to scrape by and avoid the reanimated corpses. It is now up to a small group of scientists working tirelessly in the lab to find a possible cure for the epidemic and ceasing of the contagion.

Out of part luck, strength, wits, and solid collaboration with your partner, so far you have managed to survive. You were together in a software engineering conference, when the speaker was assaulted by the reanimated, and spread panic accross the room. It has been 3 years since your first encounter with the reanimated dead, and you have formed a group which might just be able to bring an end to this mess.

You've grouped up with a few other scientists, which believe that they may just be able to extract a cure, or at least a zombification represant, that could salvage the remnants of humanity. They believe they can do this by altering the virus to aerosol form, and perform tests on it. However, this is the drawback — this exposes them into a high risk environment.

This is where you, and your possible partner come together (if your partner has not been bitten yet that is), and lay down the specification for a safety critical room. The room should be able to detect if the airborne version of the virus has been released in the room, and perform certain actions, to ensure the lives of the scientists are not endangered.

Your system affects the lives of the scientists. Your system fails, the scientists die. The scientists die, there is no cure. No cure, humanity extinct. You are the last hope humanity has — make it happen.

1.1 Foreword about concurrency

We show concurrency in EFSMs, by performing a fork, very similar to the if statement, only that the symbol is a black bar. The 'Fork' black bar accepts from one side one directed edge. On the other side, we may have many outgoing edges, which describes a concurrent process for each. All these concurrent processes should eventually join back into another bar — the 'Join' bar. The 'Join' bar accepts many directed edges on one side, and has only one outgoing edge on the other. When joining back together, each thread should have some event or guard, to mark this transition. For this assignment, if there is a concurrent part, please separate each concurrent process in a different region. Please refer to Figure 1 for a graphical representation.

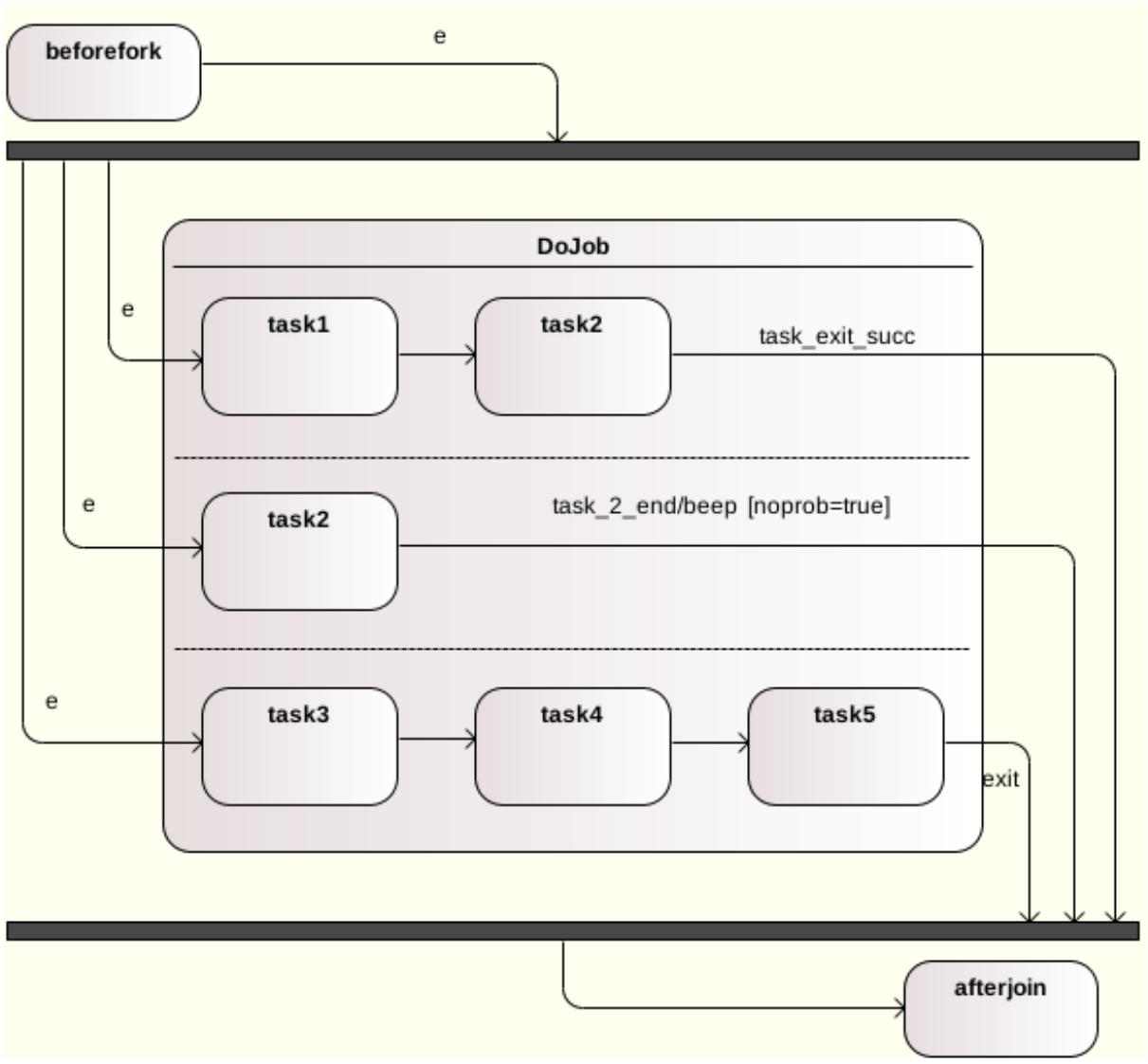

Figure 1: Example of addressing concurrency in an EFSM: 1 of 2.

For this assignment, you do not need to treat the `DoJob` state as a refined state when

writing the specification down. Follow the same syntax as you would, if you were treating an 'if' statement: by writing both execution paths inside your transition set 'Λ'. For example if we write down the transitions of `beforefork`, to `task2`, to `afterjoin`, we would get the following:

$$
\begin{aligned}
&\Lambda = \{ \\
&\quad beforefork \xrightarrow{\text{e}} task2, \\
&\quad task2 \xrightarrow{task\_2\_end[noprob=true]/beep} afterjoin \\
&\quad beforefork \xrightarrow{\text{e}} task1 \\
&\quad task1 \xrightarrow{\cdots} task2 \\
&\quad task2 \xrightarrow{task\_exit\_succ} afterjoin \\
&\quad \ldots \\
&\}
\end{aligned}
$$

Another reminder about refining. For this assignment, if you have a refinement in the form displayed in Figure 2, then you can write the specification for the transition in the following way, for the outside part:

$$
\begin{aligned}
&\Lambda_{unrefined} = \{ \\
&\quad A \xrightarrow{\text{e}} B \\
&\}
\end{aligned}
$$

And later on we can write the specification of the refined part in the following way — notice that we can show the starting state with a single right arrow; you don't need to write down the origin of the transition:

$$
\begin{aligned}
&\Lambda_{refined} = \{ \\
&\quad \rightarrow C \\
&\quad C \xrightarrow{\text{e}_2} D \\
&\}
\end{aligned}
$$

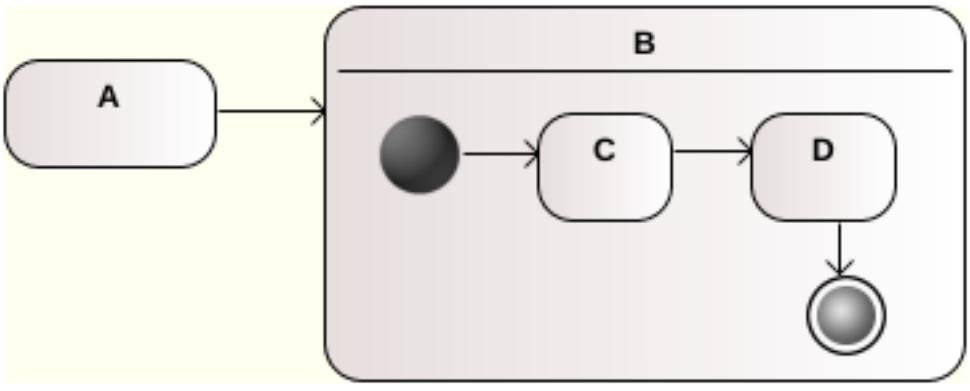

Figure 2: Example of addressing concurrency in an EFSM: 2 of 2.

2 Initial system

We will start off with a simple overview of the system. As we go on with the specification, you will be expected to refine the state transition diagram.

> **Note**: Provide a diagram *for each* part of this assignment. It is advised you make copies of the diagrams as you refine them, to retain a history. You are required to provide *all* diagrams of *each* step. Therefore 5 diagrams. If you do not provide these diagrams, you *will* be penalized.

Part I: Overall EFSM

Our overall system requires a few measures to operate in the desired fashion. We want it to begin in a `dormant` state — our system should never really be off. But if there is need of a killswitch, then an event `kill`, may rend the system to the final state.

We wish to have the system boot up, and load appropriate drivers for the hardware we are using. We transition to the state `init` from `dormant`, given we evoke a `start` event.

Once the `init` state makes sure that all the drivers are loaded, then it is ready to transition to the next state. It does so by a `init_ok` event, and transitions to the `idle` state.

The next step, since all the required drivers have been loaded, and the hardware has not caused any errors, is to enter into the `monitoring` state. Essentially this is where the scientists are inside the room and conducting experiments for the cure. To transition to this state from the `idle` state, the `begin_monitoring` event is caught.

> **Error Handling**: Many things can go wrong on a running system. Errors will be experienced. But given different scenarios, how do we deal with these errors? In systems, such as ours, which cannot afford to be turned off, crash, or panic, we need some sort of mechanism. This mechanism will try to apply different protocols to save the system from completely panicking, given the situation. This mechanism will be portrayed in the `error_diagnosis` state.

Having said that, we will now describe transitions from the previous 3 states `dormant`, `init`, and `idle`, to the error handling state.

init errs: it could be the case that during booting, and loading appropriate kernel modules or drivers, that something goes wrong. Your system should attempt to reboot if that's the case. To save the system from unlimited reboots if booting is always causing errors, we set a limit of 3. We can count these using a counter variable called `retry`. If there is an error, the event `init_crash` is caught, and the system enters the `error_diagnosis` state. If this happens, this information must be logged. A `init_err_msg` is broadcasted in this case.

The error is caught in the `error_diagnosis` state, and error saving protocols are applied. Now, the system is ready to attempt once more to boot. Given that the number of tries have not exceeded 3 attempts, a transition back to `init` is made possible, via the event `retry_init`. This is also where the counter is incremented.

safe_shutdown: given the above description, we want to prevent the system into an endless cycle of failed boot attempts. To do this we set the counter for use. Once the counter's value is or exceeds 3, we perform a graceful shutdown, which makes sure to clean up the previous state of the system, and not cause any more problems down the line. The event `shutdown` helps us transition to the `safe_shutdown` state. Finally, the `safe_shutdown` state, transitions via the event `sleep` back to the `dormant` state.

idle errs: The idle state might throw an error. Same action is preferred if that happens: try to apply the error protocol, and return to the `idle` state. The transition of `idle` to `error_diagnosis` is made possible via the event `idle_crash`. An error message is broadcasted if this crash happens. This is portrayed by the action `idle_err_msg`. The error handling happens in `error_diagnosis`, and there is a transition back to `idle`, given an event `idle_rescue`.

monitoring errs: Very similarly, when the `monitor` state experiences some sort of error, it will transist to the `error_diagnosis` state via the event `monitor_crash`. There will be another message broadcasted via an action `moni_err_msg`. The error protocols are applied, and from `error_diagnosis`, we transition back to `monitoring` via a `moni_rescue` event.

Part II: Refine init

We now proceed to refine the `init` state. Our system has to first boot some basic resource management components. Then it will sequentially try to load the kernel modules for each sensor we have. We have a temperature, and pressure sensor.

The first state we enter to, inside this refined state is the `boot_hw` state, that loads any hardware modules that are needed to bootstrap the system. Given by the event `hw_ok` denoting that everything was ok in `boot_hw`, we enter the next state `senchk`.

Next, the sensors are checked within `senchk`. If everything is ok, the event `senok` triggers the transition to the `tchk` state. This is the temperature sensor.

Next, we want to check for the pressure sensors. Given that we everything was okay in `tchk`, a transition happens to `psichk` given the event `t_ok`.

Finally, given that everything is fine with loading the pressure kernel modules, the event `psi_ok` is fired, and we end up in the state `ready`.

Part III: Refine monitor

In the monitoring state, our scientists are inside the room, where they are converting the virus to aerosol form and attempting to find a cure. We need to take measures to prevent contagion to the rest of the facility, and the scientists if this happens. We refine this state to take safety into account.

We begin in the `monidle` state. Every unit of time, we are checking if there is a contagion. In the case of no contagion, the event `no_contagion` is received, and we transition to the state

regulate_environment. After 100 milliseconds, the event after_100ms is received, and we transition back to the monidle state. On the other hand, if potential contagion is detected, then the event contagion_alert is received, and we are brought to the state lockdown. We also need to broadcast a message to the whole facility called FACILITY_CRIT_MESG. Inside the lockdown state, there will be different trials to try and purge the potential contagion. When this success is a reality, the event purge_succ is fired, and from the lockdown, we may return to the state monidle.

There is one last part to take care of. Inside refined states, recall that we can use any transition from the super state. This implies the possibility that we may exit the lockdown and monitoring state via its transitions. We want to restrict this by providing a inlockdown variable of type Boolean. If a contagion is detected, we wish no such transitions to be possible, until the purge is finally successful. This restriction is set and unset, before and after the state lockdown. You must place. In your diagram, place the extra guard that would prevent a transition outside of monitoring. You do not need to update the previous specification of Part II. Provide only the form of the new transition element, as it would be if it would be updated. In this new diagram for this particular part, add this guard as well.

Part IV: Refine lockdown

We now wish to refine the lockdown state. What happens here is that conducting a few experiments, the scientists have found out that contagions may be surpressed by alternating temperature and pressure in high frequencies, inside a controlled environment. In other words, in the case of a contagion alert, we can control the pressure and temperature of the room to stop the aerosol from infecting the scientists.

Inside the lockdown state, we begin in the state prep_vpurge. The event initiate_purge helps us transition to the next states. We also want to make sure that the doors are locked, so the action lock_doors should be sent as well in this transition. After prep_vpurge, we wish to perform the actions of the next states at the same time. The two following states are the temperature and pressure controllers, used to alternate high-low temperature and pressure values within the room. These states are alt_temp, and alt_psi. These states perform the alterations, until an event of completion for each is received. One is tcyc_comp, for the temperature, and the other is psicyc_comp for the pressure. Once both of these processes are complete, we go to the next state risk_assess, a state that gathers information, and assesses the risk. Now perform a check against a risk variable. The check dictates that if the *risk* is greater than 1%, then the purge is to restart from the beginning. If the risk is less than 1%, then the system transitions to the state safe_status. An action is fired to unlock the doors, since the purge was successful. This action is called unlock_doors. From the safe_status state, we may reach a final state.

Part V: Refine error diagnosis (robust error handling)

Finally we refine the error_diagnosis state. Once we enter the refined error_diagnosis state, we are initially in the error_rcv state. The system checks the error, and searches for

an appropriate protocol to address the error with. This check is done against a `Boolean` variable `err_protocol_def`. If the outcome is `true`, then the system enters a state called `applicable_rescue`. The event `apply_protocol_rescues` is fired, and we may finish up by transitioning to the final state. On the other hand, if no rescue protocols have been found, we will simply reset the kernel module data. Therefore on a `false` case, we enter the state `reset_module_data`. An event is fired up called `reset_to_stable` to confirm that the reset has been successful. This also ends up in the final state.

Part VI: Simulating your machine

Adopt the fact structure that you covered in class (see lecture notes on *Simulating automata* by the instructor) and transform your EFSM into a declarative database. Proceed to extend the database with the following rules:

1. `is_loop(Event, Guard)` succeeds by finding a loop edge. We assume that an edge can be represented by a non-null event-guard pair.
2. `all_loops(Set)` succeeds by returning a set of all loop edges.
3. `is_edge(Event, Guard)` succeeds by finding an edge.
4. `size(Length)` succeeds by returning the size of the entire EFSM (given by the number of its edges).
5. `is_link(Event, Guard)` succeeds by finding a link edge.
6. Rule `all_superstates(Set)` succeeds by finding all superstates in the EFSM.
7. `ancestor(Ancestor, Descendant)` is a utility rule that succeeds by returning an ancestor to a given state.
8. `inheritss_transitions(State, List)` succeeds by returning all transitions inherited by a given state.
9. `all_states(L)` succeeds by returning a list of all states.
10. `all_init_states(L)` succeeds by returning a list of all starting states.
11. `get_starting_state(State)` succeeds by returning the top-level starting state.
12. `state_is_reflexive(State)` succeeds is `State` is reflexive.
13. `graph_is_reflexive` succeeds if the entire EFSM is reflexive.
14. `get_guards(Ret)` succeeds by returning a set of all guards.
15. `get_events(Ret)` succeeds by returning a set of all events.
16. `get_actions(Ret)` succeeds by returning a set of all actions.

17. `get_only_guarded(Ret)` succeeds by returning state pairs that are associated by guards only.

18. `legal_events_of(State, L)` succeeds by returning all legal event-guard pairs.

3 Guidelines

For this assignment you must work in pairs. You must work strictly between the two of you and you may not seek assistance on this assignment from anyone, including the instructor or the Teaching Assistants. You will only receive feedback upon submission. Further guidelines are discussed in the subsequent subsections.

3.1 Version control

Use git SCM for version control. Whenever you do some work, commit your changes with appropriate messages. Before submitting, run the git garbage collector via 'git gc' (make sure you don't want to rescue anything because git gc removes old references/backups). When submitting the assignment, make sure you include the .git subfolder (the repository) along with your assignment. You can obtain more information at the following link: `http://git-scm.com/docs/gittutorial`.

3.2 What to submit

You will have to submit five specifications (with their corresponding state transition diagrams). In your specifications, do not omit empty sets. Do not include the state transition diagrams inside the specification document. The reason is so that you can freely use a lot of space to make your diagrams, and not be constrained by the length or width of the page. Please provide the diagrams in preferably `PNG` format. Formats such as `JPG`, and `GIF` are fine too.

Additionally, you will need to provide your code, for the prolog implementation. We recommend two files: one would be your `fact file`, where you store your `Prolog Database`. The other file shall be your `rule file` which stores any queries that you are required to implement for the assignment.
Your should package all your files in a zip file. Your zip file should have an 8-digit name that is the concatenation of each partner's id's last 4 digits. You should electronically submit the zip file under *Theory Assignment 3*.

3.3 Technology recommendations

You can use any software to build state transition diagrams. A recommended tool, with a slight learning curve, which is provided for free is called `Modelio`. You may get the software in the following link: `http://www.modelio.com`.

Note Well for Modelio: You'll need to copy the EFSM diagram each time you are extending it. If you extend the diagram in the same space, you will not retain previous versions of the diagram. To do this, simply right click on the diagram in question, and select 'copy'. Then, right click on the project folder, and select 'paste'.

For `Prolog` there exists two interpreters that you may use: `SWI Prolog (swipl)`, and `Gnu Prolog (gprolog)`. Regardless what you use, please state it clearly in your assignment. Our recommendation is to use `swipl`: `http://www.swi-prolog.org`.

3.4 Due date and late submissions

Your assignment is due by March 30 at 21:00. Any late submission by one day will get a 50% penalty and it will subsequently receive a 10% penalty per day.

3.5 Disputes

If you wish to dispute the grade that you receive on your assignment, please email me, providing all details. I will discuss your dispute with the Teaching Assistants and respond to your email. My decision in my response will be final.