3815ICT-Software Engineering Workshop 8

**Activity 1**

Reactive-systems are responsive systems without much processing; their counterparts are deliberative systems (which show artificial intelligence capabilities such as reasoning, planning, and learning) [1]. A Graphical User Interface (GUI) in common desktop operating systems or tablets is perhaps a good example of a reactive system that is not a real-time system: many users have experienced how the “mouse” turns into a spinning icon of a sand-clock for an indefinite amount of time. Find an alternative scholar reference (citation) that supports this definition of a *reactive system.*

**Response 1**

https://patents.google.com/patent/US9298266B2/en

**Activity 2**

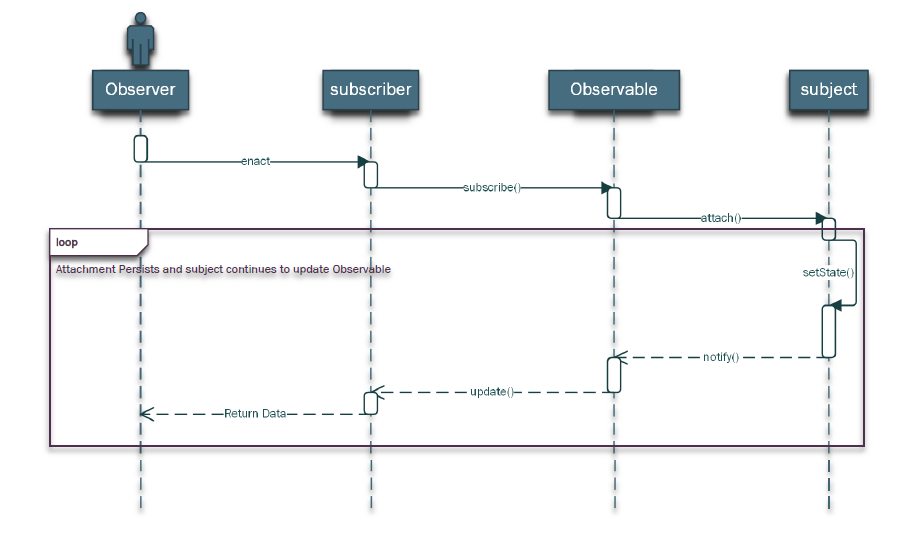
A GUI is also an example of an event-driven system; namely one typically based on a software architecture built around stimuli-driven callbacks, a subscribe mechanism, and listeners that enact such call-backs. Reacting to stimuli in this way implies uncontrolled concurrency (e.g. using separate threads or event queues). Another example of an eventdriven system is the interrupt mechanism for device I/O. Lamport [4] provided fundamental proofs of the limitations of event-driven systems. The counterpart to an event-driven system is a time-triggered system. The counterpart of the interrupt mechanism is polling. Study the class diagram and the sequence diagram for the Observer Pattern in Figure 1. (en.wikipedia.org/wiki/Observer\_pattern). Note that the scenario presented is similar to an arrival of an event to a listener who has several subscribers. However, in this case

1. instead of registering a call-back, the first thing that happens is that observers attach themselves to a subject.

2. later, when the setState() method is invoked on a subject, the subject executes its notify() method which in itself invokes the update() method in each attached observer.

Using this analogy, draw a sequence diagram of the subscription mechanism and the use of callbacks.

**Response 2**



**Activity 3**

*Real-time systems* are required to meet time-deadlines in response to stimuli [3]. Therefore, although closely related, these three types of systems are not the same. The ubiquity of state charts to model these three types of systems has blurred the distinctions; for example, Sommerville [6, p. 544], refers to UML as an illustration that “state models are often used to describe real-time systems”. However, Lamport [4] has provided solid and persuasive arguments to why real-time systems may be better served by time-triggered systems and pre-determined schedules, rather than the unbounded delays that may occur in event-driven systems.

Find an alternative scholar reference (citation) that supports that *real-time systems* are not exactly the same as *reactive system* and that also are not exactly the same as *event-driven systems.*

**Response 3**

https://patents.google.com/patent/EP1503329A2/en?q=real+time+systems&q=reactive+systems&q=event+driven+systems

**Activity 4**

Consider the UML illustrative statechart in Figure 2. In Activity 6 of Workshop 3 you were asked to review that UML’s state diagrams (which is a derivation of Harel’s statecharts [2]) are event-driven. You were asked to study the notation in the wikipedia website that shows this diagram.

• en.wikipedia.org/wiki/UML\_state\_machine

This figure also appears in Samek’s book on executable UML’s statecharts [5]. The UML statechart was also used in the midterm of this course in 2018. Make a list of the events this model of a Toaster Oven is supposed to react to. Whiteboard/Blackboard Middleware.

**Response 4**

* DOOR\_OPEN
* DOOR\_CLOSE
* DO\_TOASTING
* DO\_BAKING

**Activity 5**

The Robotic Operating System (ROS) is a middleware that offers two fundamental patters.

1. Publisher/Subscriber (see Figure 3b).

2. Client/Server (see Figure 3a).

We will use these two patterns in making and executable model of Figure 2 with LogicLabelled Finite-State Machines. You will be provided with a *catkin* workspace for this workshop. In there there is a package called *event\_wrapper*. You are welcome to perform the *ROS* beginners tutorials if you want to learn more about the middleware infrastructure of *ROS*, but this is not necessary. This activity consist on compiling all packages in the workspace provided and then executing the node *event\_wrapper*, then using ROS tools to observe the communication patterns. To compile the packages, inside the directory catkin\_ws you launch the package manager.

*catkin\_make*

This will produce two executables for the package event\_wrapper in the folder devel/lib. After compiling, you can register communications in the ROS environment executing

*source devel/setup.bash*

In a separate terminal run the middleware than enables the communication

*roscore*

You can terminate when finish running ROS programs the middleware with a kill signal such as Ctrl-C. You run the main one as follows if in the directory of the workspace:

*./devel/lib/event\_wrapper/event\_wrapper\_node*

This node records when someone posts an event in the corresponding topic. The was to generate an event (from another terminal) is as follows.

*rostopic pub /events\_on\_toaster std\_msgs/String toast*

Do not worry if you do not understand everything that is here, but this is our current way of generating an event to toast. There are other 3 events:

*rostopic pub /events\_on\_toaster std\_msgs/String bake*

*rostopic pub /events\_on\_toaster std\_msgs/String close*

*rostopic pub /events\_on\_toaster std\_msgs/String open*

Issue these events from a separate terminal. You need to exit rosopic with Ctl-C. What is the effect of these events. They get recorded for a client to inspect. You can simulate a client with

*rosservice call /what\_door\_event\_recorded*

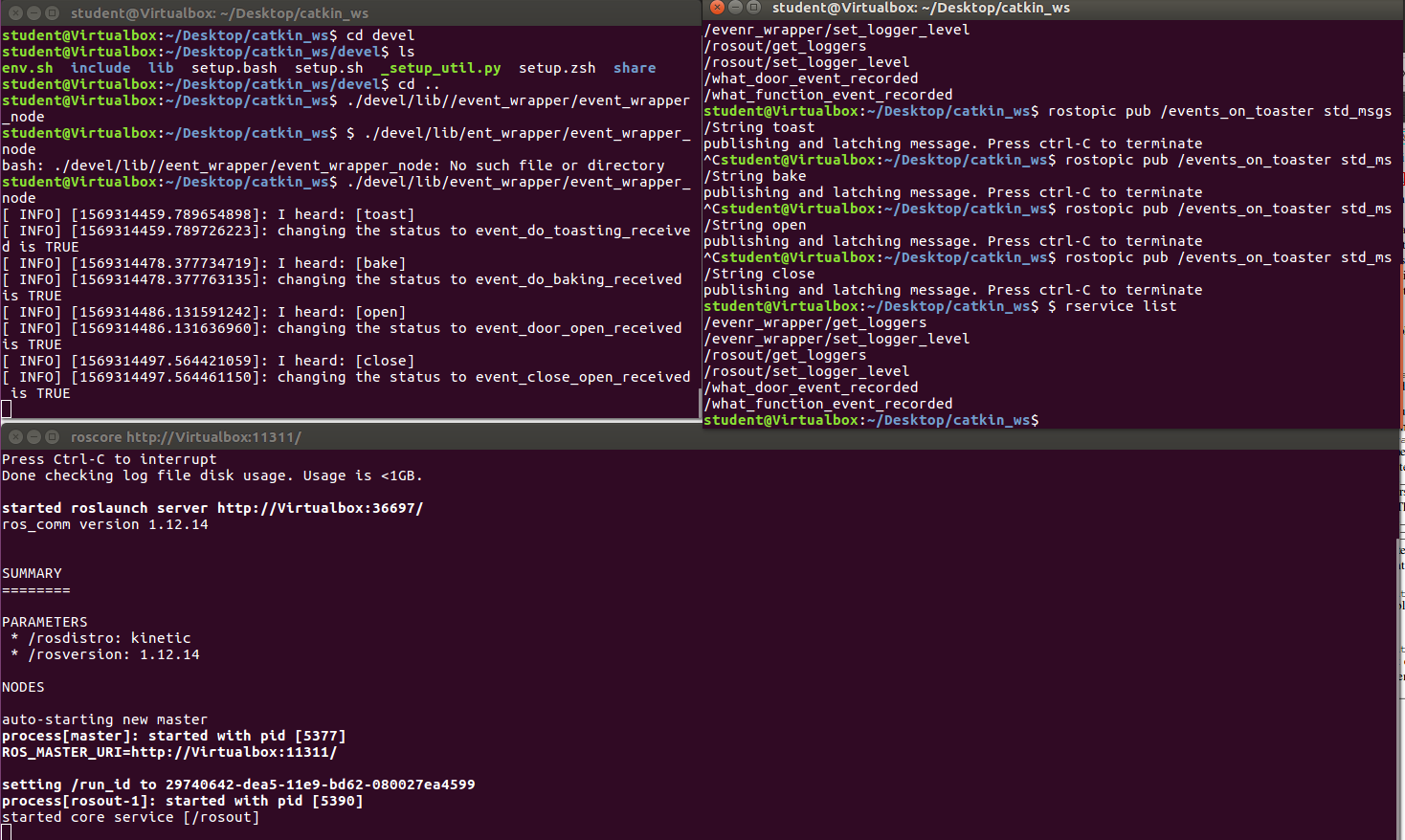
This is the first service, to be able to inquiry what events regarding the door have been received since your last inquiry. Similarly there is another service, which you query as follows:

*rosservice call /what\_function\_event\_recorded*

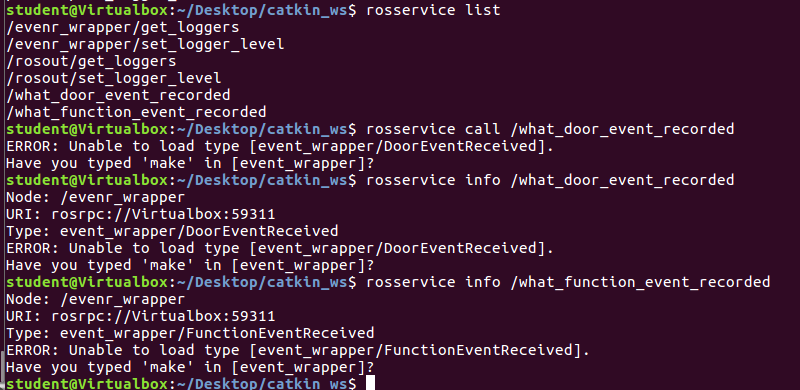
This service lets you know what events regarding toasting or baking have been received since your last inquiry. Experiment with posting several events and then query the events received in the two services.

**Response 5**

First figure shows an image of initial Catkin environment up and running along with basic commands being issued



On attempted execution of the rosservice calls, I received the following errors consistently and was unable to progress any further.



Attempts to remedy this problem were unsuccessful.

**Activity 6**

In this activity we will execute the two executable models that implement the UML diagram in Figure 2. The first llfsm appears in Figure 4a and control switching between toasting and baking. Your task is to use the previous activity (Activity 5) to post event suitable for this behaviour once it is executing. Although Activity 5 compiled this machine already (you can inspect devel/lib in the workspace and you will see libInner.so), you provably need the path for the scheduler of *llfsms*(the scheduler is called *clfsm*). Thus, go to where the source code of the machine is and set it up.

*cd src/Inner/machine*

*./machine\_catkin\_setup.sh Inner.machine*

You need to compile the machine, so at the root of the workspace issue

*catkin\_make*

Then go back to devel/lib and copy it as *clfsm* expect it.

*cp libInner.so Inner.machine/Linux-x86\_64/Inner.so*

In another terminal run *roscore*, in a second one run the *event\_wrapper* and in a third one under devel/lib run the model.

*./clfsm/clfsm Inner*

From another fourth terminal publish events as discussed in Activity 5. Observe how the llfsm emits messages to controller. Compare the output with the design in Figure 2.

Stop the machine. You may need to find its process id with *ps -x* and issue an explicit *kill* command with the process id. Execute it in verbose mode

*./clfsm/clfsm -v Inner*

and compare the result with Figure 4a.

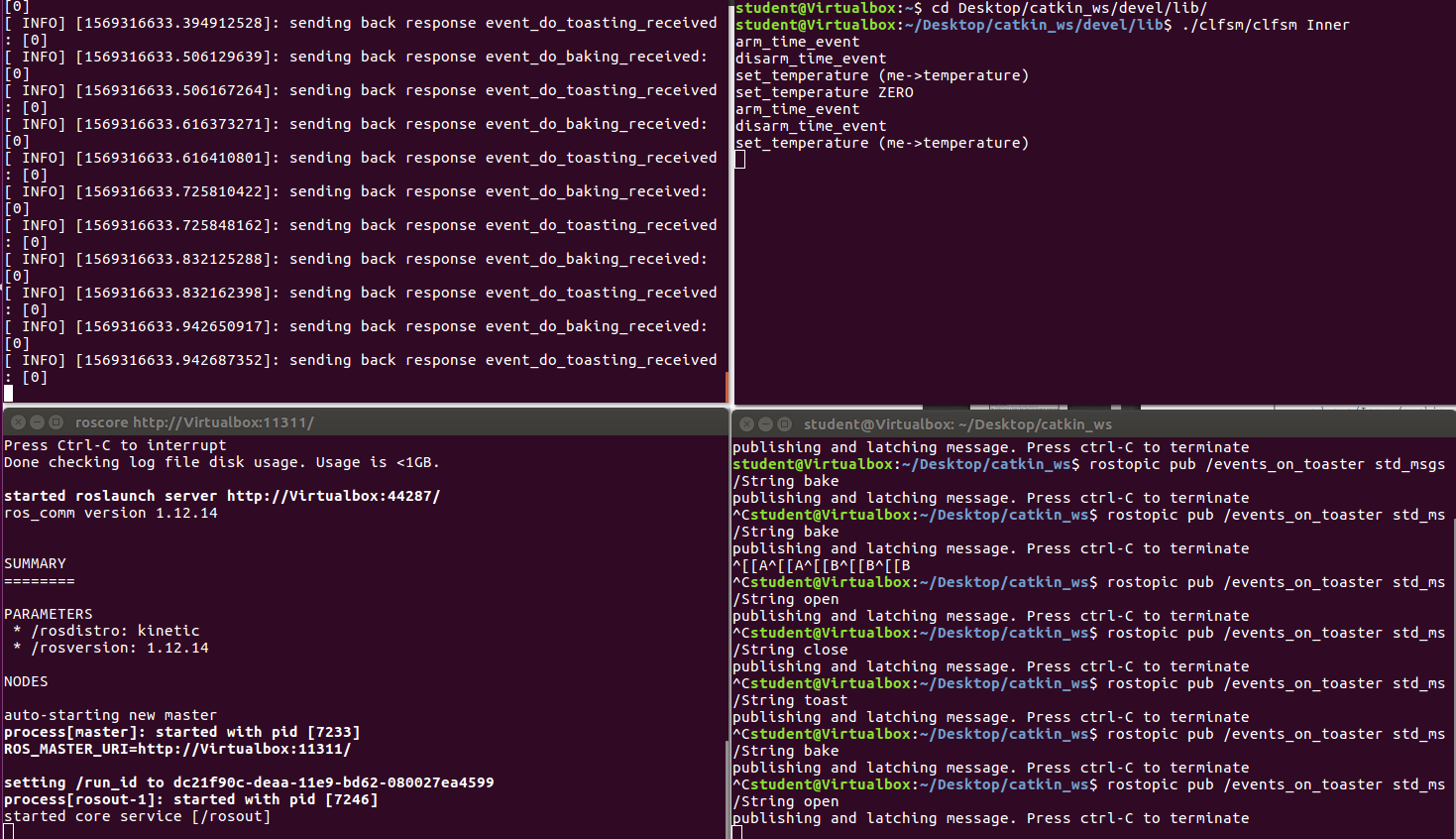
The second *llfsm* is illustrated in Figure 4b. Repeat the process for the *Outer* machine. That is run its setup and copy it into the path of *clfsm* before you execute it. Then issue events as describe in Activity 5. Again compare with the design in Figure 2. Run it in verbose mode and compare it with Figure 4b. Finally, execute both *llfsms* together.

*./clfsm/clfsm Outer Inner*

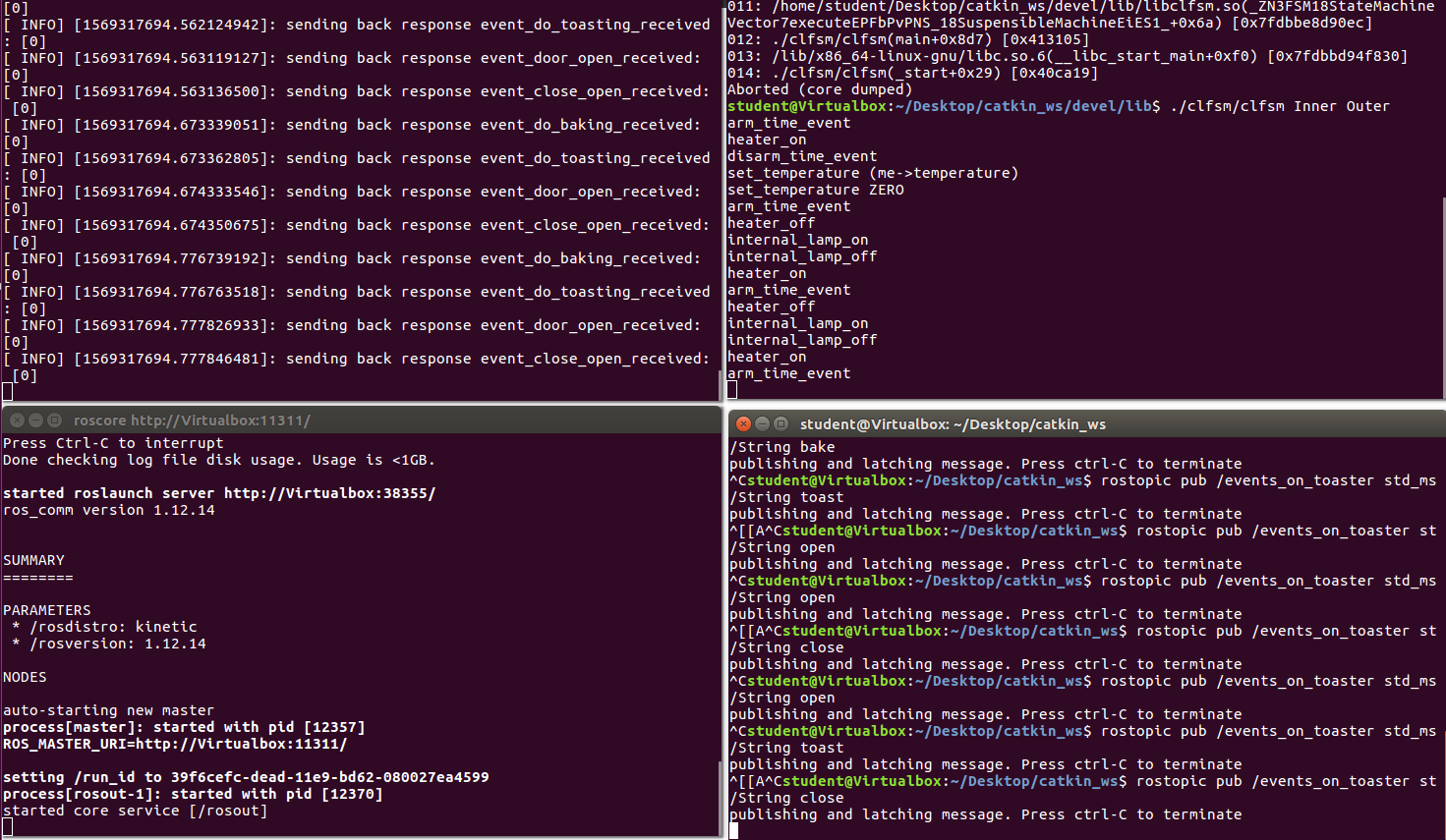
Issue events about the doors and also abuot whether toasting or baking. Compare with the design in Figure 2. Evaluate how well does the implementation fit the design.

**Response 6**

The first step of running clfsm Inner has worked, it appears to allow only the Inner Machine to hold functionality. When I attempt to call open or close, the machine continues to run bake or toast. When I transition between Bake & Toast, everything appears to work as intended.



Conversely, compiling and running the Outer Machine allowed it to register the open and close commands which allowed for entry and exit between the two Outer states. It did not allow any use of the Inner states of bake/toast. It can only be deduced that running the two together will allow for full functionality of the Machine.



The implementation seems to fit the figure for the Logic Labelled State Machine very well. The machine is unable to run toast or bake while the door is opened and the door must be closed before they are able to run. Furthermore, if commands such as bake are issued from the Inner machine while the Outer machine is still running, they will be read by the scheduler and discarded. They will not be stored to run upon the Door\_closed event. Aside from this, the functions appear to run their internal functions upon activation of their respective events and the entry / exit events run as designated by the model.

**Activity 7**

Write 15 lines of a reflective report on the previous activities. Analyse and evaluate the match of the activities to the learning objectives proposed in this workshop/laboratory.

**Response 7**

The learning objectives for this workshop were first of all – to distinguish between reactive, event-driven and real-time systems. In order to do this, an exercise delved into observers and observables. I feel like this particular objective was well-met, however I found the scholarly articles quite dry and learn most of my information from youtube videos. I feel there was a disconnect between some of these objectives, for instance model driven development and design patterns. I generally was not aware that I was dealing with design patterns and particularly a mode of development. It frustrates me sometimes when I am learning abstract ideas in order to apply them to my programming when I have already been applying them organically without any knowledge of the framework itself. I get confused by the jargon when I technically already understand the concept as I’ve applied it before, it isn’t until it clicks that I say… “ohhhh… I’ve been doing that all along anyway…”. As for UML diagrams and understanding the relevance of models, I definitely understand this and I quite enjoy reading models and seeing their relevance in the big picture of a project. I believe this objective has been met.