17-651 Models of Software Systems
Project 2: Concurrency

Group 9

Tianli Wu

Vishwas Singh

Wei-Hsuan Wang

Yanhao Li

Yuanzong Zhang

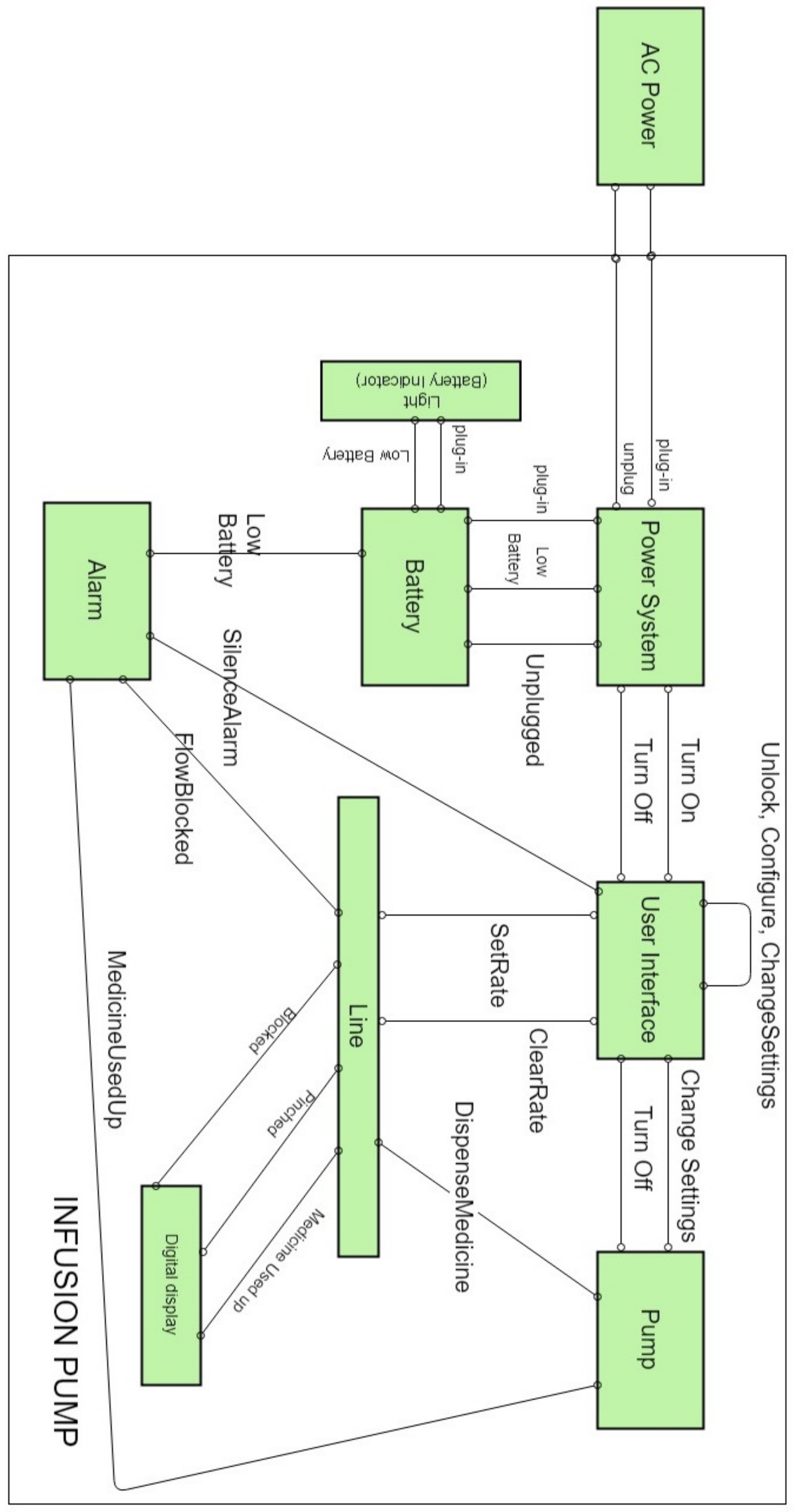
November 16, 2018

In the project, we produced an infusion pump that includes a power system(POWER), an alarm(ALARM), an user interface(UNIT), and a low-battery light(LOWBATTERYLIGHT). The pump contains two infusion lines, each includes a line mode(LINE), an infusion mode(INFUSION), and an error digit that shows what error happens to the infusion line. The error digit is a unique number that corresponding to a specific action that either triggers an alarm or handles it. These numbers will be displayed on a screen together with the low battery light. The low battery light will be turned on when the system battery is low and turned off when the battery is recharged. To simplify the system, we abstracted the system and omitted the detailed amount of medicine and pumping rate. We show only the action of setting rate while do not pass in any actual numbers for the setting action. Another thing that worth notice is that we made the finish pumping to be an action instead of a state. The detailed assumptions and design decisions we made are shown in the following section.

Assumptions

1. The power failure and manually unplug the power line are the same action. (both switch the power source from AC power to the battery)
2. The power recover and manually plug in the power line are the same action. (both switch the power source from the battery to AC power)
3. When the power line is plugged in, the battery can be changed automatically.
4. The lines are physically connected to the pump. The user can take actions to the lines at any time no matter the pumo is turned on or turned off.
5. The user can erase and unlock line only when the line is locked.
6. The user can confirm settings to start pumping only when the line is locked.
7. The user should take the connect set, purge air, lock line actions orderly to a line.
8. The flow can be blocked at any time when pumping has started.
9. The user can only take the unblock the flow action to the line when the flow is blocked.
10. The line can be pinched at any time when pumping has started.
11. The user can only take the unpinch the line action to the line when the line is pinched.
12. The user can only take the change medicine action to the line when the the medicine is used up.
13. The alarm will be activated when the flow is blocked, the medicine is used up, the line is pinched, or the battery is low.
14. The alarm will be activated after the pump is tunred on if the flow is blocked or the medicine is used up or the line is pinched when the pump is not turned on.
15. The pump will not dispense flow unless the line is locked, the rate and value are set and the control unit is locked.
16. When confirm the settings to a line, the control unit can be automatically locked. (to start dispense)
17. The alarm can be scilenced no matter the control unit is locked or not.
18. The pump can be manually turned off only when the control unit is unlocked.
19. When the pump is turned on, the error digit will display the error type of the line until the error is handled.
20. When the pump is turned on, the low-battery will display when the battery is the power source and it has low energy.

BLOCK DIAGRAM



Note : This block diagram contains only limited essential transitions and actions.

Task 1: Modeling with Concurrency

The FSP code has been attached at the end of the document.

Task 2

1. The pump cannot start pumping without the operator first confirming the settings on the pump.
   (a) This property is a safety property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the assertion feature in LTSA to check this property.

```
assert SETBEFOREDISPENSE =
    forall[lineIndex:LineIndexT]
        (!dispense_main_med_flow[lineIndex] U confirm_settings[lineIndex])

No LTL Property violations detected.
```

2. Electrical power can fail at any time.
   (a) This property is a liveness property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the fluent and assertion features in LTSA to check this property.

```
fluent ELEON = <{plug_in}, {unplug}>
assert ELECTRICALFAILANYTIME =
    [](ELEON -> []<>unplug)

No LTL Property violations detected.
```

3. If the backup battery power fails, pumping will not occur on any line.
   (a) This property is a safety property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the fluent and assertion features in LTSA to check this property.

```
fluent BATTERYUDERUP = <{battery_used_up}, {plug_in}>
assert NODISPENSEWITHDEADBATTERY =
    [](forall[lineIndex:LineIndexT] (BATTERYUDERUP ->
        (!dispense_main_med_flow[lineIndex] U plug_in)))

No LTL Property violations detected.
```

4. It is always possible to resume pumping after a failure.
   (a) This property is a liveness property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the fluent and assertion features in LTSA to check this property.

```
fluent POWERFAIL =
    <{unplug}, {plug_in}>
assert RESUMEDISPENSEPOSSIBLE =
    [](forall[lineIndex:LineIndexT] (POWERFAIL ->
        <>(confirm_setting -> dispense_main_med_flow[lineIndex])))

No LTL Property violations detected.
```

5. An alarm will sound on any line failure (blockage, pinching, empty fluid, or whatever failures you model).
   (a) This property is a liveness property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the fluent and assertion features in LTSA to check this property.

```
fluent ALARMON[lineIndex:LineIndexT] =
    <{flow_blocked[lineIndex], medicine_used_up[lineIndex], line_pinched[lineIndex]},
        {silence_alarm, turn_off, battery_used_up}>
assert ALARMWILLSOUND =
    [](forall[lineIndex:LineIndexT] (ALARMON[lineIndex] -> <>silence_alarm))

No LTL Property violations detected.
```

6. In the absence of errors the pump will continue to pump until the treatment is finished.
   (a) This property is a liveness property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the fluent and assertion features in LTSA to check this property.

```
fluent FLOWERROR[lineIndex:LineIndexT] =
    <{flow_blocked[lineIndex]}, {flow_unblocked[lineIndex]}>
fluent MEDICINEERROR[lineIndex:LineIndexT] =
    <{medicine_used_up[lineIndex]}, {change_medicine[lineIndex]}>
fluent PINCHERROR[lineIndex:LineIndexT] =
    <{line_pinched[lineIndex]}, {line_unpinched[lineIndex]}>
assert PUMPWILLFINISHIFNOERROR =
    [] (forall[lineIndex:LineIndexT]
        (([]!FLOWERROR[lineIndex] && []!MEDICINEERROR[lineIndex] && []!PINCHERROR[lineIndex]) ->
            <>main_med_flow_finish[lineIndex]))

No LTL Property violations detected.
```

7. The system never deadlocks.
   (a) This property is a safety property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the safety check feature provided by LTSA to check this property.

```
No deadlocks/errors
```

8. Property A: Battery cannot be used up until power fail and the battery has become low.
   (a) This property is a safety property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the fluent and assertion features in LTSA to check this property.

```
fluent POWERFAIL =
    <{unplug}, {plug_in}>
fluent BATTERYLOW =
    <{low_battery}, {plug_in}>
assert BATTERYUSEUP =
    [](!battery_used_up U (POWERFAIL && BATTERYLOW))

No LTL Property violations detected.
```

9. Property B: When the keypad is locked, the pump cannot be turned off manually.
   (a) This property is a safety property.
   (b) Our model allows us to check this property.
   (c) The check result is true. We use the fluent and assertion features in LTSA to check this property.

```
fluent LOCKSCREEN =
    <{lock_unit, confirm_settings[LineIndexT]}, {unlock_unit, battery_used_up}>
assert NOMANUALTURNOFFWHENLOCKUNIT =
    [](!turn_off U !LOCKSCREEN)

No LTL Property violations detected.
```

Task 3

1. What are the strengths of this notation and its tools? Under what situations would you recommend its use? Why?

   - Pre-post condition:
     - ▷ Strength: The pre-post condition is powerful in separating the precondition and the postcondition of an operation.
       Alloy, does not have concept of precondition or postcondition to be defined explicitly, and everything is treated as a constraint. Alloy evaluates each statement and tries to find instances that satisfy all of them. There is no way that we can explicitly separate preconditions and postconditions.
       In LTSA, the only possible way that we can express pre-post condition is to use the keyword "when," which states that some action can happen only when a particular condition is true. However, this method can only efficiently express preconditions and postconditions with few constraints. When they require many criteria to be true at the same time, LTSA cannot express them in a clear and easy way. On the other hand, the pre-post condition can express preconditions and postconditions separately and clearly. All constraints can be placed together by using logical operators, no matter how much constraints we have. In addition, preconditions and postconditions can be easily distinguished as they are written separately.
     - ▷ Recommended usage: The situation that we would recommend to use pre-post condition is when we need to express operations or simple models. The pre-post condition shows the effect of an operation clearly as it explicitly states the situation before the operation and the situation after the operation, Therefore, it is a great way to express operations. Besides, as it is much easier to understand and use than Alloy and FSP; it is a good option for simple models that only involves few operations and objects.

- Alloy:
  - ▷ Strength: Alloy is good at expressing the relationship between different objects as well as the interactions among them. The relationship can be expressed by the concept of inheritance. For example, we can express "student" as a subset of "human" in Alloy. Every object in Alloy is represented as a signature, and all actions must happen between signatures. This makes the interaction between each object extremely clear. Because of its object-oriented nature, Alloy is similar to a programming language, which means alloy models can be easily translated into Java or C++ code. On the other hand, FSP specifies everything as a state and does not distinguish objects while the pre-post condition is only good at represent operations in a single system.
  - ▷ Recommended usage: The situation that we would recommend to use Alloy is when the target system can be easily divided into separate objects or subsystems. In this case, the interactions between systems are usually more important than the events flow. Alloy can easily express every individual subsystem as a signature and its tasks as operations(methods), which is what we want. Moreover, it can show how all the subsystems work together to achieve the functionality of the entire system.
- FSP (LTSA):
  - ▷ Strength: FSP is powerful in expressing the events flow of a system and how many systems synchronize with each other. In both Alloy and pre-post condition, we can hardly express an action flow that contains a sequence of actions. However, in LTSA, as every action is represented as a state change, it is easy to express a sequence of actions. Also, LTSA can handle concurrency well, because it can express a sequence of operations. Concurrency is the coordination between multiple systems, and a single action will not involve concurrency. This is why Alloy and the pre-post condition is bad at representing concurrency. LTSA, on the other hand, expresses model using sequences of actions and states, which is what needed for representing concurrency.
  - ▷ Recommended usage: The situation that we would recommend to use FSP is when we need to handle concurrency between different systems. In this case, the execution order will be more important than the interaction between systems as wrong order might lead to deadlock. LTSA is a process-oriented tool, which use states to express the action flow. This provides the possibility of representing concurrency, as the execution order can be controlled by changing the action flow in different order.

2. What are the weaknesses of this notation and its tools. Under what situations would you not recommend its use? Why?

- Pre-post condition:
  - ▷ Weakness: The pre-post condition notation has two essential shortcomings. The first one is that it can not model complex systems because of its limited expression ability. The pre-post condition can represent the effect of an operation well, but it models a system through the representation of the operations inside that system. This way can work for the simple system. However, for a complex system, many properties cannot be expressed only by operations. They will require more powerful expressions like invariants or properties. Another weakness of pre-post condition is that it does not have an analyzer. Without an analyzer, all the interpretation need to be done by hands, which is very time consuming and inefficient.
  - ▷ Situation to avoid: The situation that we would suggest not to use pre-post condition is when we want to model a large system. An extensive system usually contains some hidden requirement that can not be expressed by operations. In this case, we will need to restrict the system behavior in some way other than the operations. However, the pre-post condition does not support complicated descriptions such as constraints in Alloy or properties in LTSA. Therefore, the pre-post condition might not able to fully account for all features of the system we are trying to model.
- Alloy:
  - ▷ Weakness: Alloy also has two significant disadvantages. The first one is that Alloy proves a model's correctness by trying to find counterexamples of it, which is not a good way. Many incorrect modeling of the system can also lead to the situation of no counterexample found. A

typical case can be when there is no valid instance at all. In this case, Alloy won't find any counterexample while the model is actually wrong. The counterexample check can be misleading in this circumstance. Furthermore, Alloy cannot express concurrency as it is not good at expressing sequences of operations. Single operation between systems is not enough to show concurrency.

  - ▷ Situation to avoid: The situation that we would suggest not to use Alloy is when we need to show the action flow of the system, especially when we need to synchronize several systems. Alloy is an object-oriented tool, which emphasizes the relationship and interactions between systems rather than the process within the system. Although Alloy can have multiple states, the model will be extremely complicated if we need to represent many states in the model.

- FSP (LTSA):
  - ▷ Weakness: The weakness of FSP is that it can not represent entity relationships. Every system in the FSP world is a separate process. We can easily analyze each process or synchronize different processes. However, it is hard to express that one process belong to another one. Specifically, there is no concept of inheritance or extend in LTSA, which indicates it can not express entity relationship clearly. There is another weakness particular for the LTSA tool, which is it can only generate images with limited transitions. When we try to debug the model in LTSA, we usually look at the generated graphs as they are much more straightforward than using the buttons. Meanwhile, the graph cannot be generated when the model gets complex, and the transitions are too much to be displayed on a single screen.
  - ▷ Situation to avoid: The situation that we would recommend not to use FSP is when we need to do a complex object modeling. Object modeling usually involves entity relationships and focus more on the interactions instead of transitions. In this case, as the LTSA tool can not express entity relationship, it might not be a good choice. In addition, LTSA treats every object as an individual process, and the objects can only be connected through synchronization. Therefore, LTSA cannot express the interactions between systems clearly, which is an essential part in object modeling.

3. With respect to this notation, what is the single most-important future development that would be needed to make it more generally useful to practitioners?

- Pre-post condition:
  - ▷ The most important future development for the pre-post condition is that it needs to have an analyzer. Currently, we write and evaluate the pre-post condition statements all by hands, which strongly limited the power of the notation. Our brain can not interpret large and complex system, and that is why we need computers to do the advanced analysis. Therefore, when the system gets completed, it is not reasonable to expect that any people can analyze the model without using any tool. On the other hand, if a system is easy enough to be understood without the help of an analyzer, we do not need a model of that system. Hence, if a notation does not has any supporting tool, it is not likely that the notation can be used in industry as it can only be used to model small systems.
- Alloy:
  - ▷ We think the most important future development for Alloy is that it can provide a complementary tool to help transfer the UML class diagram into Alloy code. Alloy represents objects with separate signatures, and it has the concept of inheritance, which makes it pretty much like an object-oriented programming language. Like we usually need to draw a UML diagram to design a program in Java or C++, we might need to do the same for Alloy. In this case, it would be really neat if we can use some tool to directly transfer our UML model into the Alloy code so that we do not need to spend extra time doing it ourselves. So far, there are many tools that can create class structures for Java or C++ based on the UML diagram, and some of them can do it pretty well. Considering the similarity between Alloy and modern object-oriented programming, we believe it won't be too hard to implement this type of translator.
- FSP (LTSA):
  - ▷ For FSP, we believe the most important future development can be a Java code generator based on LTSA code. The FSP notation using states to represent sequences of actions, which is the

essential reason why it can support concurrency modeling. This is similar to how Java deal with concurrency. Actually, when we were doing the homework, all the models provided in the textbook has an equivalent version of Java code. It is not so hard to transform LTSA code into Java code while it is still time-consuming. Therefore, it would be a good idea to let a generator do this for us. Considering the task the LTSA is handling concurrency in a way similar to Java, the cost of developing such a tool should be in a reasonable amount.

FSP Code

```
//======================
// Constants and Ranges
//======================

//
// States of the pump alarm
//
const AlarmActivated = 0    // Alarm currently active
const AlarmSilenced  = 1    // Alarm currently inactive

range AlarmStateT = AlarmActivated .. AlarmSilenced

//
// States of the pump settings
//
const ParamsNotSet = 2    // pump parameters not set yet
const ParamsSet    = 3    // pump parameters already set

range ParamsStateT = ParamsNotSet .. ParamsSet

//
// Locked/unlocked states of a line with respect to a pump channel
//
const LineUnlocked = 4  // line not locked into a pump channel
const LineLocked   = 5  // line locked into a pump channel

range LineLockStateT = LineUnlocked .. LineLocked

//
// Locked/unlocked states of the pump unit
//
const UnitUnlocked = 6  // the keypad of the pump is not locked
const UnitLocked   = 7  // the keypad of the pump is locked

range UnitLockStateT = UnitUnlocked .. UnitLocked

//
// Line indexes
//
const LineMinIndex = 8
const LineMaxIndex = 9

range LineIndexT = LineMinIndex .. LineMaxIndex

//
// States of Power
```

```
//
const PowerOn = 10
const PowerOff = 11
range PowerStateT = PowerOn .. PowerOff

//
// Sources of Power
//
const PowerSourceAC = 12
const PowerSourceBattery = 13
const PowerSourceBatteryLow = 14
const PowerSourceNone = 15
range PowerSourceT = PowerSourceAC .. PowerSourceNone

//
// Error status
//
const NoLineError = 0
const FlowBlockError = 1
const LinePinchError = 2
const MedicineUsedUpError = 4
const AllError = FlowBlockError + LinePinchError + MedicineUsedUpError
range ErrorStateT = NoLineError .. AllError

//
// Display status
//
const DisplayOff = 16
const DisplayOn = 17
range DisplayStateT = DisplayOff .. DisplayOn

//
// Low battery light status
//
const LowBatteryLightOff = 18
const LowBatteryLightOn = 19
range LowBatteryLightStateT = LowBatteryLightOff .. LowBatteryLightOn

//
// Set of actions that the user of the LTSA tool can control in an
// animation of this model.
//
menu UserControlMenu = {
    change_settings[LineIndexT], clear_rate[LineIndexT],
    confirm_settings[LineIndexT], connect_set[LineIndexT],
    dispense_main_med_flow[LineIndexT], enter_value[LineIndexT],
    erase_and_unlock_line[LineIndexT], flow_unblocked[LineIndexT],
    sound_alarm, lock_unit, plug_in, press_cancel[LineIndexT],
    press_set[LineIndexT], set_rate[LineIndexT], silence_alarm,
    turn_off, unlock_unit, unplug, flow_blocked[LineIndexT],
    turn_on, purge_air[LineIndexT], lock_line[LineIndexT]
}

// The power system, which supply the power to the infusion
// pump. Possible powers are battery and AC power.
POWER = POWERSYETEM[PowerOff][PowerSourceNone],
POWERSYETEM[powerState:PowerStateT][powerSource:PowerSourceT] = (
    // when the AC power is on, it can be turned off
    when (powerState == PowerOn)
        turn_off -> POWERSYETEM[PowerOff][powerSource]
    |
    // when the power is off, and there is some type of
    // source, it can be turned on
    when (powerState == PowerOff && powerSource != PowerSourceNone)
        turn_on -> POWERSYETEM[PowerOn][powerSource]
    |
    // the AC power can be unplugged
    when (powerSource == PowerSourceAC)
        unplug -> POWERSYETEM[powerState][PowerSourceBattery]
    |
    // when the AC power has not been plugged in,
    // it can be plug in
    when (powerSource != PowerSourceAC)
        plug_in -> POWERSYETEM[powerState][PowerSourceAC]
    |
    // when the system is using battery, the battery will
    // go low in some state
    when (powerState == PowerOn && powerSource == PowerSourceBattery)
        low_battery -> POWERSYETEM[powerState][PowerSourceBatteryLow]
    |
    // when the battery is low, it will dead in some state
    when (powerState == PowerOn && powerSource == PowerSourceBatteryLow)
        battery_used_up -> POWERSYETEM[PowerOff][PowerSourceNone]
).

// The individual line. An infusion pump can
// contain as many lines as we want (yet only two
// in this project). Each individual line is working
// separately, it has no interactions with other lines.
LINE(LineIndex=LineMinIndex) = LINESTATE[LineUnlocked],
LINESTATE[lineLock:LineLockStateT] = (
    // before the line is locked, the operator
    // need to connect it and purge air
    when (lineLock == LineUnlocked)
        connect_set[LineIndex] -> purge_air[LineIndex] ->
            lock_line[LineIndex] -> LINESTATE[LineLocked]
    |
    // when the line is locked, it can be unlocked
    when (lineLock == LineLocked)
        erase_and_unlock_line[LineIndex] -> LINESTATE[LineUnlocked]
    |
    // when the line is locked, the operator can
    // confirm settings and start pumping
    when (lineLock == LineLocked)
        confirm_settings[LineIndex] -> LINEPUMP
),

LINEPUMP = (
    // during pumping, the operator can change setting
    // at any time
    change_settings[LineIndex] -> LINESTATE[LineLocked]
    |
    // during the dispensing of medicine, the diepensing
    // might finish successfully, be interrupted, and
    // the medicine might go empty
    dispense_main_med_flow[LineIndex] -> (
        dispense_not_finsihed[LineIndex] -> LINEPUMP
        |
        main_med_flow_finish[LineIndex] -> LINESTATE[LineLocked]
    )
    |
    medicine_used_up[LineIndex] -> change_medicine[LineIndex] -> LINEPUMP
    |
    // the line can be pinched or blocked during pumping
    flow_blocked[LineIndex] -> flow_unblocked[LineIndex] -> LINEPUMP
    |
    line_pinched[LineIndex] -> line_unpinched[LineIndex] -> LINEPUMP
).

// The Infusion pump, it can contain as many lines
// as we want(yet only two in this project). It also
// synchronize with alarm and user interface.
INFUSION(LineIndex=LineMinIndex) = INFUSIONOFF[ParamsNotSet],
INFUSIONOFF[params:ParamsStateT] = (
    // when the infusion is off, it can be turned on
    turn_on -> INFUSIONSETUP[ParamsNotSet]
),

INFUSIONSETUP[params:ParamsStateT] = (
    // when the parameter is not set, the
    // operator can set it
    when (params == ParamsNotSet)
        press_set[LineIndex] -> INFUSIONSETUP[ParamsSet]
    |
    // when the parameter is set, it can be reset
    when (params == ParamsSet)
        clear_rate[LineIndex] -> INFUSIONSETUP[ParamsNotSet]
    |
    // when the parameter is set, the operator
    // can confirm settings and start pumping
    when (params == ParamsSet)
        confirm_settings[LineIndex] -> INFUSIONPUMP
    |
    // at any time the pump can be turned off
    turn_off -> INFUSIONOFF[params]
    |
    // at any time the battery might be used up
    battery_used_up -> INFUSIONOFF[params]
),

INFUSIONPUMP = (
    // during pumping, the operator can change
    // the settings at any time
    change_settings[LineIndex] -> INFUSIONSETUP[ParamsSet]
    |
    // during pumping, the medicine might be used up
    medicine_used_up[LineIndex] -> INFUSIONSETUP[ParamsSet]
    |
    // if nothing else happen, the pump will
    // dispense medicine
    dispense_main_med_flow[LineIndex] -> INFUSIONPUMP
    |
    // the pump will finish when the needed amount is dispensed
    main_med_flow_finish[LineIndex] -> INFUSIONSETUP[ParamsNotSet]
    |
    // at any time the pump can be turned off
    turn_off -> INFUSIONOFF[ParamsSet]
    |
    // at any time the battery might be used up
    battery_used_up -> INFUSIONOFF[ParamsSet]
).

// The alarm system, which will trigger an alarm
// when bad situations happen. There are four possible
// alarms in our model, which are low battery alarm,
// block line alarm, no medicine alarm, and line pinched
// alarm. All alarms can be manually silenced through
// the user interface
ALARM = ALARM_OFF[AlarmSilenced],
ALARM_OFF[alarmState:AlarmStateT] = (
    // when the alarm is off, it can be turned on
    turn_on -> ALARM_STATE[alarmState]
    |
    // when the line is blocked, the block line alarm
    // will be triggered
    flow_blocked[lineIndex:LineIndexT] -> ALARM_OFF[AlarmActivated]
    |
    // when the medicine is used up, the no medicine alarm
    // will be triggered
    medicine_used_up[lineIndex:LineIndexT] -> ALARM_OFF[AlarmActivated]
    |
    // when the line is pinched, the line pinched alarm
    // will be triggered
    line_pinched[lineIndex:LineIndexT] -> ALARM_OFF[AlarmActivated]
),

ALARM_STATE[alarmState:AlarmStateT] =
(
    // when the alarm is activated, it can be manually silenced
    when (alarmState == AlarmActivated)
        silence_alarm -> ALARM_STATE[AlarmSilenced]
    |
    // the four actions will activate their corresponding alarms
    flow_blocked[lineIndex:LineIndexT] -> ALARM_STATE[AlarmActivated]
    |
    medicine_used_up[lineIndex:LineIndexT] -> ALARM_STATE[AlarmActivated]
    |
    line_pinched[lineIndex:LineIndexT] -> ALARM_STATE[AlarmActivated]
    |
    low_battery -> ALARM_STATE[AlarmActivated]
    |
    // at any time the alarm can be turned off
    turn_off -> ALARM_OFF[AlarmSilenced]
    |
    // at any time the battery can be used up
    battery_used_up -> ALARM_OFF[alarmState]
).

// The user interface of the system. The operator
// can change and confirm settings here. The operator
// can not do anything once the keypad is locked
UNIT = UNITOFF,
UNITOFF = (
    // when the UI is off, it can be turned on
    turn_on -> UNITSTATE[UnitUnlocked]
),

UNITSTATE[unitLock:UnitLockStateT]= (
    // when the keypad is locked, it can be unlocked
    when (unitLock == UnitLocked)
        unlock_unit -> UNITSTATE[UnitUnlocked]
    |
    // when the keypad is not locked, it can be locked
    when (unitLock == UnitUnlocked)
        lock_unit -> UNITSTATE[UnitLocked]
    |
    // the pump can start pumping only after the
    // keypad is locked
    when (unitLock == UnitLocked)
        dispense_main_med_flow[lineIndex:LineIndexT] -> UNITSTATE[UnitLocked]
    |
    // when the keypad is not locked, the operator
    // can set the pumping rate
    when (unitLock == UnitUnlocked)
        set_rate[lineIndex:LineIndexT] -> (
            // the battery might used up during setting
            battery_used_up -> UNITOFF
            |
            enter_value[lineIndex] -> (
                // the battery might ussed up during entering value
                battery_used_up -> UNITOFF
                |
                // the user can set value
                press_set[lineIndex] -> UNITSTATE[UnitUnlocked]
                |
                // the user can cancel setting
                press_cancel[lineIndex] -> UNITSTATE[UnitUnlocked]
            )
        )
    |
    // when the keypad is not locked, the operator can
    // confirm settings
    when (unitLock == UnitUnlocked)
        confirm_settings[lineIndex:LineIndexT] -> UNITSTATE[UnitLocked]
    |
    // when the keypad is not locked, the operator can
    // change settings
    when (unitLock == UnitUnlocked)
        change_settings[lineIndex:LineIndexT] -> UNITSTATE[UnitUnlocked]
    |
    // when the keypad is not locked, the operator can
    // reset the rate
    when (unitLock == UnitUnlocked)
        clear_rate[lineIndex:LineIndexT] -> UNITSTATE[UnitUnlocked]
    |
    // the operator can manually silence the alarm
    // through UI
    silence_alarm -> UNITSTATE[unitLock]
    |
    // when the keypad is not locked, the system can be
    // turned off
    when (unitLock == UnitUnlocked)
        turn_off -> UNITOFF
    |
    // the battery might used up
    battery_used_up -> UNITOFF
).

// The error message display used to inform the
// operator which type of alarm has occured as well
// as if the alarm has been handled
ERRORDIGIT(LineIndex=LineMinIndex) = ERRORDIGITSTATE[DisplayOff][NoLineError],
ERRORDIGITSTATE[display:DisplayStateT][errorState:ErrorStateT] = (
    when (display == DisplayOff)
        // when the display system is off, it can be turned on
        turn_on -> ERRORDIGITSTATE[DisplayOn][errorState]
    |
    // Different actions will trigger different messages
    flow_blocked[LineIndex] ->
        ERRORDIGITSTATE[display][errorState | FlowBlockError]
    |
    flow_unblocked[LineIndex] ->
        ERRORDIGITSTATE[display][errorState & (AllError - FlowBlockError)]
    |
    line_pinched[LineIndex] ->
        ERRORDIGITSTATE[display][errorState | LinePinchError]
    |
    line_unpinched[LineIndex] ->
        ERRORDIGITSTATE[display][errorState & (AllError - FlowBlockError)]
    |
    medicine_used_up[LineIndex] ->
        ERRORDIGITSTATE[display][errorState | MedicineUsedUpError]
    |
    change_medicine[LineIndex] ->
```

```
        ERRORDIGITSTATE[display][errorState & (AllError - MedicineUsedUpError)]
    |
    when (display == DisplayOn)
        // at any time the system can be turned off
        turn_off -> ERRORDIGITSTATE[DisplayOff][errorState]
    |
    when (display == DisplayOn)
        // at any time the battery might be used up
        battery_used_up -> ERRORDIGITSTATE[DisplayOff][errorState]
).


// The light used to inform the operator that the
// battery is low. It can be displayed only when the
// display system is on
LOWBATTERYLIGHT = LOWBATTERYLIGHTSTATE[DisplayOff][LowBatteryLightOff],
LOWBATTERYLIGHTSTATE[display:DisplayStateT][lowBatteryState:LowBatteryLightStateT] = (
    when (display == DisplayOff)
        // when the display system is off, it can be turned on
        turn_on -> LOWBATTERYLIGHTSTATE[DisplayOn][lowBatteryState]
    |
    // when the AC power is plugged in, the light will turn off
    plug_in -> LOWBATTERYLIGHTSTATE[display][LowBatteryLightOff]
    |
    // when the battery is low, the light will turn on
    low_battery -> LOWBATTERYLIGHTSTATE[display][LowBatteryLightOn]
    |
    when (display == DisplayOn)
        // when the display system is on, it can be turned off at any time
        turn_off -> LOWBATTERYLIGHTSTATE[DisplayOff][lowBatteryState]
    |
    when (display == DisplayOn)
        // the battery might be used up at any time
        battery_used_up -> LOWBATTERYLIGHTSTATE[DisplayOff][lowBatteryState]
).

// all systems run concurrently
||PUMP = (POWER || forall [index:LineIndexT] LINE(index) ||
    forall [index:LineIndexT] INFUSION(index) || ALARM || UNIT ||
    forall [index:LineIndexT] ERRORDIGIT(index) || LOWBATTERYLIGHT).

// 1. The pump cannot start pumping without the operator first confirming the settings on the pump.
assert SETBEFOREDISPENSE =
    forall[lineIndex:LineIndexT]
        (!dispense_main_med_flow[lineIndex] U confirm_settings[lineIndex])

// 2. Electrical power can fail at any time.
fluent ELEON = <{plug_in}, {unplug}>
assert ELECTRICALFAILANYTIME =
    [](ELEON -> []<>unplug)

// 3. If the backup battery power fails, pumping will not occur on any line.
fluent BATTERYUDERUP = <{battery_used_up}, {plug_in}>
assert NODISPENSEWITHDEADBATTERY =
    [](forall[lineIndex:LineIndexT] (BATTERYUDERUP ->
        (!dispense_main_med_flow[lineIndex] U plug_in)))

// 4. It is always possible to resume pumping after a failure.
fluent POWERFAIL =
    <{unplug}, {plug_in}>
assert RESUMEDISPENSEPOSSIBLE =
    [](forall[lineIndex:LineIndexT] (POWERFAIL ->
        <>(confirm_setting -> dispense_main_med_flow[lineIndex])))

// 5. An alarm will sound on any line failure
fluent ALARMON[lineIndex:LineIndexT] =
    <{flow_blocked[lineIndex], medicine_used_up[lineIndex], line_pinched[lineIndex]},
        {silence_alarm, turn_off, battery_used_up}>
assert ALARMWILLSOUND =
    [](forall[lineIndex:LineIndexT] (ALARMON[lineIndex] -> <>silence_alarm))

// 6. In the absence of errors the pump will continue to pump until the treatment is finished.
fluent FLOWERROR[lineIndex:LineIndexT] =
    <{flow_blocked[lineIndex]}, {flow_unblocked[lineIndex]}>
fluent MEDICINEERROR[lineIndex:LineIndexT] =
    <{medicine_used_up[lineIndex]}, {change_medicine[lineIndex]}>
fluent PINCHERROR[lineIndex:LineIndexT] =
    <{line_pinched[lineIndex]}, {line_unpinched[lineIndex]}>
assert PUMPWILLFINISHIFNOERROR =
    [] (forall[lineIndex:LineIndexT]
        (([]!FLOWERROR[lineIndex] && []!MEDICINEERROR[lineIndex] && []!PINCHERROR[lineIndex]) ->
            <>main_med_flow_finish[lineIndex]))

// 8. Battery cannot be used up until power fail and the battery has become low
fluent BATTERYLOW =
    <{low_battery}, {plug_in}>
assert BATTERYUSEUP =
    [](!battery_used_up U (POWERFAIL && BATTERYLOW))

// 9. When the unit is locked, the pump cannot be turned off manually
fluent LOCKSCREEN =
    <{lock_unit, confirm_settings[LineIndexT]}, {unlock_unit, battery_used_up}>
assert NOMANUALTURNOFFWHENLOCKUNIT =
    [](!turn_off U !LOCKSCREEN)
```