A Coloured Petri Net Approach to Model and Analyze Safety-Critical Interactive Systems

Sapna Jaidka  
Department of Computer Science  
University of Waikato and  
Waikato Institute of Technology  
Hamilton, New Zealand  
sapnajaidka87@gmail.com

Steve Reeves  
Department of Computer Science  
University of Waikato  
Hamilton, New Zealand  
stever@waikato.ac.nz

Judy Bowen  
Department of Computer Science  
University of Waikato  
Hamilton, New Zealand  
judy.bowen@waikato.ac.nz

Abstract—To gain confidence in safety-critical interactive systems, formal modelling and analysis plays a vital role. The aim of this paper is to use Coloured Petri Nets to model and analyze safety-critical interactive systems. We present a technique to construct a single Coloured Petri Net model of the user interface, interaction and functionality of safety-critical interactive systems and then analyze the achieved Coloured Petri Net model using a state space analysis method. There are several reasons for using Coloured Petri Nets. Coloured Petri Nets provides a graphical representation and hierarchical structuring mechanism, and a state space verification technique, which allows querying the state space to investigate behaviours of a system. There are several tools that supports Coloured Petri Nets including the CPN Tool which helps in building CPN models and allows simulation and analysis using state spaces. The technique to model and analyze safety-critical interactive systems is illustrated using a simplified infusion pump example.

Index Terms—Coloured Petri Nets, Formal Modelling and analysis

I. INTRODUCTION

Interactive systems are systems which have a significant amount of interaction between computers and humans. The two main components of interactive systems are the interactive and the functional. A good user-interface plays a significant role for both the components. A user interface must be simple and easy to understand so that users can complete their intended task efficiently and with minimal difficulty. User interface is made up of widgets which could be buttons, sensors, touch-screen or speech. Each widget has a certain set of behaviours associated with it which control and change the interactive and functional behaviour of the system. The functional aspect of the system deals with the operations, calculations and information it takes and returns to the user interface. There are various interactive systems which, if they fail, can cause significant damage to property, the environment or even human life. Such interactive systems are referred to as safety-critical interactive systems.

Due to advances in science and technology, there has been an increase in complexity in the safety-critical interactive systems, and this complexity can be a source of errors. There are various reasons for the failure of safety-critical interactive systems, like faulty design and software errors [1]. There could be mistakes in the specification of the system, mistakes in the software or hardware design or errors could occur when humans interact with a system. Researchers have been working for many years to solve problems or issues in safety-critical interactive systems due to poor user interfaces and functionality, for example, in avionics [2] or in medical infusion pumps [3] [4] to ensure that safety-critical interactive systems are safe before entering the market and function correctly in all situations. Therefore, it is necessary to model and analyze safety-critical interactive systems using formal methods in terms of user interface, interaction and functional specifications to ensure that the system meets all requirements.

The focus of this paper is to develop a technique to model and analyze safety-critical interactive systems using formal methods that covers all the various aspects (user interface, interaction and functional) to make sure that the system will act as expected before implemented. For modelling we take advantage of existing formalisms: Z, Presentation Interaction Models (PIMs) and Presentation Models (PMs) (as for instance in [5] [6]). From this existing basis we create a Coloured Petri Net model of safety-critical interactive systems and analyze the model using a state space method.

II. RELATED WORK

There are two strands of our research: formal methods and human computer interaction (HCI). We will look at how both of these strands are used together, with a specific focus on modelling and analyzing safety-critical interactive systems.

Formal methods have been used for HCI for many years now. This is evident by looking at Jacky’s work in [7] where Z [8] has been used for modelling the interface of a radiation machine. Paterno et al. in [9] used LOTOS for modelling and evaluating the usability of user interfaces. Doherty and Harrison in [10] used VDM for describing interactors and formal reasoning about the functionality of a user interface. ICO [11] is one more example of a formalism that was developed based on the existing formalism of Petri Nets. Barboni et al. in [12] used ICO on a cockpit display system for describing the interactive widgets, user applications and user interface servers. Thimbleby in [13] has presented work to detect errors in safety-critical interactive systems related to number entry. Silva et al. [14] reverse engineer an abstract
model of a graphical user interface and ensure it satisfies the set of requirements. They have also described the IVY project which is used for analysis of interactive system design. Campos and Harrison in [15] have used the IVY tool with Modal Action Logic (MAL) for modelling and analysis of interactive systems. Researchers have also worked on creating models that can be used in the design process of interactive systems in a more general manner to ensure the requirements are met, for example, [16].

These general modelling approaches have to tackle the problem of separation of concerns between interface and functional elements, while at the same time managing the relationship between the two [17] [18]. One of the focuses of this paper is to address this issue and bridge the gap between the user interface and underlying system functionality.

III. FORMAL METHODS AND TOOLS

Formal Methods are mathematically-based techniques, languages and notations that are used for designing, modelling and analyzing hardware and software systems. The main advantage of formally modelling a system is that we assure that our design really does what is intended and we are able to identify potential problems before we can implement a system.

In our work we have used existing formalisms: presentation models, presentation interaction models and Z (PM/PIM/Z) as our basis [6]. These three formalisms result in three separate models (user interface, interaction and functional) and a lot of work is required to do the coupling of these separate models [18]. To overcome this issue, we use Coloured Petri Nets (CPN) and create a single model which have all the three aspects (user interface, interaction and functional).

Coloured Petri Nets [19] is a language used for the modelling and analysis of systems. There are a lot of tools that can be used for CPN [20] [21]. CPN Tool v.4.0.1 from [22] is used in this work. Using CPN Tool, it is possible to investigate the behaviour of the modelled system using simulation, to verify properties by means of state space methods and model checking by writing functions in SML. These are the main reason of choosing CPNs. We illustrate our technique with a simplified infusion pump example.

IV. EXAMPLE: SIMPLIFIED INFUSION PUMP

As the focus of this paper is on safety-critical interactive systems, we consider the simplified infusion pump as shown in Figure 1 to help explain the process. It comprises of one Display and six buttons; Display, NoButton, YesButton, InfoButton, MinusButton, PlusButton and OnOffButton button. This makes total of seven widgets.

There is a total of six states, one of which the pump can be in at any given point in time, which are: init, info, setvolume, settieme, confirmrate and infusing. Init is the initial state of a system. Then the user can press press InfoButton and can see the battery information. A user can then set the time and volume using the PlusButton and MinusButton and lastly press YesButton to start infusing.

There is a set of operations/functions associated with the simple infusion pump like IncreaseVolume, DecreaseVolume, SetVolume, IncreaseTime, DecreaseTime, SetTime, SetRate and StartInfusing. In each state there are seven observations that can be made: battery whose value can be sufficient or insufficient, infusing whose value can be yes or no, timer, volume, volumeleft, infusionrate and timeleft are the observations that have natural number as their value. In the init state, we assume that the battery is sufficient, pump is not infusing and all the other observations are set to zero. The operations IncreaseVolume and DecreaseVolume allow the user to change the value of a dose to be infused using the PlusButton and MinusButton. These operations changes the value of the observation volume by adding one or subtracting one from its current value. The operation SetVolume will set the value of the observation volumeleft to the current value of the observation volume. The operations IncreaseTime and DecreaseTime allow the user to change the duration of infusion using the PlusButton and MinusButton. These operations changes the value of the observation timer by adding one or subtracting one from its current value. The operation SetTime will set the value of the observation timeleft to the current value of the observation timer. The SetRate operation will set the the value of the observation infusionrate by dividing the current value of volume by the current value of timer and the operation StartInfusing will set the value of the observation infusing to yes.

V. MODELLING SAFETY-CRITICAL INTERACTIVE SYSTEMS USING COLOURED PETRI NETS

We start with describing a process [23] to model user interface using Coloured Petri Nets with the help of simple infusion pump example as shown in Figure 1. For modelling a user interface of an interactive system, three things that needs consideration are: firstly, all the widgets that a system have, secondly, the category of these widgets, i.e., deciding if the widget is just a responder which displays messages or the widget is actually doing some action etc., and thirdly what type of behaviours are associated with these widgets, i.e., functions that are executed when the widget is interacted with. These could be underlying non-interactive system functions which we call S-behaviours and interactive functions which
we call I-behaviours. Each widget is represented as a triple 
"(WidgetName, Category, (Behaviour(s)))" and each state of a
system is described separately as a set of triples. For example,
there are seven widgets and six states for simple infusion
pump, then each state is represented as a set of seven triples.

The user interface of a simple infusion pump is given in
Table I. WidgetName is an enumeration type that represents
the names of the seven widgets (Display, NoButton, YesButton,
InfoButton, MinusButton, PlusButton and OnOffButton) of a
simple infusion pump shown in Figure 1. Each of these
widgets are categorized using the widget categorization hi-

Table II shows the colour sets (types) and variables which
are required to form the arc expressions to model function-
ality of a system. Colour set INFUSING is declared as the
enumerated colour set that can have exactly two values Yes or
No. Colour set BATTERY is also declared as an enumerated
colour set that can have two values sufficient or insufficient.
Colour set NAT is declared as the integer colour set. Colour
set Z is a record colour set with a record of all the seven
observations. As the operations/functions would be written as
arc expressions so we need to declare variables which could be
bound to different values of their respective colour sets during
simulation. There are seven variables battery, timer, volume,
volumeleft, timeleft, infusionrate and infusioning as shown in
Table II.

Figure 2 shows the structure of the Init page of the pump
which models the init state. There are two places init and info
which belong to init and info fusion sets respectively. This
page shows the navigational possibility from the init state. It
clearly shows that from init we can move to the info state by
firing the transition I_info. If we look at the init constant in
Table I, there is only one I-behaviour, so the model will
contain just one transition. A marking on the place init shows
the value of the constant init which gives information about
the available widgets and their associated behaviours in the
init state. As each place has the colour set pmodel attached
to it, so the arc expression must also be of the same type.
The constant init forms the arc expression for the arc which is
going outwards from the init place and info will form the arc
expression for the incoming arc expression of place info. We
have one fusion place named Z added to the model which is
of type colour set Z. A marking on the place Z is the record of
the observations described in section IV showing the initial
values.

Figure 3 shows the structure of page setvolume which
represents the setvolume state of a pump. This state allow
user to set the volume to be infused. The model comprises
of four places: setvolume, info, setttime and Z and four tran-
sitions : I_info, S_DecreaseVolume, S_IncreaseVolume and
I_settime_S_setvolume. We have named the transitions with
TABLE I

USER INTERFACE OF SIMPLE INFUSION PUMP USING COLOURED PETRI NETS

| colset WidgetName = with Display | NoButton | YesButton | InfoButton | MinusButton | PlusButton | OnOffButton; |
| colset Category = with ActCtrl | Responder; |
| colset Behaviour = with S_death | S_trap; |
| colset Behaviours = list Behaviour; |
| colset widgetdescr = product WidgetName * Category * Behaviours; |
| colset pmodel = list widgetdescr; |

val init = [(Display, Responder, [S_displaystartmessage]), (NoButton, ActCtrl, []), (YesButton, ActCtrl, [I_info]), (MinusButton, ActCtrl, [I_info]), (PlusButton, ActCtrl, []), (OnOffButton, ActCtrl, [])];

val info = [(Display, Responder, [S_displaybatterylife]), (NoButton, ActCtrl, [I_init]), (YesButton, ActCtrl, [I_settime]), (MinusButton, ActCtrl, []), (PlusButton, ActCtrl, []), (OnOffButton, ActCtrl, [])];

val setvolume = [(Display, Responder, [S_decreasevolume, S_increasevolume]), (NoButton, ActCtrl, [I_info]), (YesButton, ActCtrl, [I_settime, S_setvolume]), (MinusButton, ActCtrl, []), (PlusButton, ActCtrl, []), (OnOffButton, ActCtrl, [])];

val infusing = [(Display, Responder, [S_displayinfusingmessage]), (NoButton, ActCtrl, []), (YesButton, ActCtrl, []), (MinusButton, ActCtrl, []), (PlusButton, ActCtrl, []), (OnOffButton, ActCtrl, [])];

TABLE II

COLOUR SETS AND VARIABLES FOR FUNCTIONALITY

| colset INFLUSING = with Yes | No; |
| colset BATTERY = with sufficient | insufficient; |
| var battery : BATTERY; |
| var timer : NAT; |
| var volume : NAT; |
| var volumeleft : NAT; |
| var infusionrate : NAT; |
| var timeleft : NAT; |
| var infusing : INFLUSING; |

An output arc from fusion place Z to the S_increaseVolume transition has an expression that simply contains assignments which set each variable to its current value. This set of assignments "picks up" the current values of the variables ready to be used by the second arc, i.e. an input arc to the place Z from the same S_increaseVolume transition. This second arc, the one to the place Z from the same S_increaseVolume transition assigns each variable to its new value. Every time the transition S_increaseVolume fires, the value of the variable volume is increased by one.

We can also add guards to the transitions. If we want that the transition S_increaseVolume should not be enabled if the value of the observation volume is greater than or equal to three, then we can add the guard [volume < 3] to the transition S_increaseVolume as shown in Fig. 3. Similarly we model the other two operations DecreaseVolume and SetVolume by adding the relevant arc expressions and guards as shown in Fig. 3.

Because of the space restrictions we are not showing the L_settime_S_setvolume as these two behaviours occur simultaneously. The model shows the navigational possibility from the setvolume state. A user can go to the settime state by firing the transition L_settime_S_setvolume or can go to the state info by firing the L_info transition. If a user is in setvolume state, then the marking on the place setvolume will have a token that will give information about the available widgets and its associated behaviour. Occurrences of transitions will update the corresponding markings on the places. There are three functions associated with the setvolume state: IncreaseVolume, DecreaseVolume and SetVolume. The arc expressions on the arcs going to and from the S-behaviour transitions represents the functionality.
info, settime and confirmrate pages. The last state is the infusing state and the CPN model of this state is shown in Fig. 4. There is no other state where the pump can go after entering the infusing state. This is how we can make use of the Coloured Petri Nets to create a single model of safety-critical interactive systems which have all the three aspects (user interface, interaction and functionality). Now we want to investigate if the CPN model is behaving as intended. In the next section we will investigate the behaviour of the CPN model of the pump.

VI. ANALYSIS AND FINDINGS

For safety-critical interactive systems, it is important to ensure that a system is behaving as expected. In this work we use a state space method to analyze the behaviour of a system. The state space graph is a set of nodes that are connected by arcs. A strongly connected component (SCC) of the state space is a maximal sub-graph, whose nodes are mutually reachable from each other [25].

State space methods have several advantages [26]: they can be automatically constructed, which provides computer-aided analysis and verification of the behaviour of the modelled system; the tool is fully integrated into the CPN Tool; it includes a lot of information about the behaviour of the system, which can answer a large set of analysis questions; it can also be used for debugging and testing the system.

A state space report can also be generated using the CPN tool [22]. The state space report includes statistical information about the size of the state space and SCC graph. In addition, the report provides information about home markings, dead markings, dead transitions and live transitions [25]. The state space report of the CPN model of simple infusion pump is summarized in table III. The complete state space comprises
of 332 nodes and 680 arcs, while the SCC graph comprises of 7 nodes and 10 arcs. The size of the SCC graph is smaller than the state space graph indicating that there are cycles. Each SCC node has particular number of nodes and arcs from state space graph, which are grouped in that SCC node.

### Table III

<table>
<thead>
<tr>
<th>Statistics of the CPN Model of Simple Infusion Pump</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State Space Graph</strong></td>
</tr>
<tr>
<td>Number of Nodes</td>
</tr>
<tr>
<td>Number of Arcs</td>
</tr>
<tr>
<td><strong>SCC Graph</strong></td>
</tr>
<tr>
<td>Number of Nodes</td>
</tr>
<tr>
<td>Number of Arcs</td>
</tr>
<tr>
<td>Home Markings</td>
</tr>
<tr>
<td>Number of Dead Markings</td>
</tr>
<tr>
<td>Dead Transition Instances</td>
</tr>
<tr>
<td>Live Transition Instances</td>
</tr>
</tbody>
</table>

#### A. Investigation of General Properties

1) **Deadlock freedom**: It is important to check if there is any deadlock in the model. Deadlock means that a user enters into a state in which no action can be taken. In CPN, a system is said to be deadlock-free if no dead marking can be reached from the initial marking. This means that from an initial state, the user will never get into a state in which no actions can be taken. A dead marking is a marking with no enabled transition. In Table III, we see our model has dead markings.

If we want to check for the improper terminations (i.e. deadlocks), we first have to find the valid terminal markings and then see in the list of dead markings if there exists a marking or markings which does not exist in the list of valid terminal markings. If there exists such marking then it means that the system has improper termination i.e. deadlock, otherwise it is free from deadlocks.

A function given in table IV detects the improper termination for our model.

#### Table IV

**Showing invalid terminal nodes for simple infusion pump model**

```haskell
fun ValidTerminalMarking n = (Mark.infuse’infuse 1 n = 1) ![Display, ActCtrl, [S_displayinfusingmessage]], ![NoButton, ActCtrl, []], ![YesButton, ActCtrl, []], ![InfoButton, ActCtrl, []], ![MinusButton, ActCtrl, []], ![PlusButton, ActCtrl, []], ![OnOffButton, ActCtrl, []])
fun InValidTerminal() = PredNodes(ListDeadMarkings(), fn n => not (ValidTerminalMarking n), NoLimit);```

The function `ValidTerminalMarking` states that marking on place `infuse` on the first instance of page `infuse` is the expected terminal marking. The function `InValidTerminal` searches through all the dead markings. If the dead marking matches the markings stated in the function `ValidTerminalMarking`, then it gives an empty list otherwise it lists the dead markings which are other than the expected terminal markings. In this case we have an empty list as an output i.e. there are no invalid terminal markings. Hence there are no deadlocks.

2) **Livelock**: Livelock i.e. a cycle in which no progress is being made, occurs when sequences are executed indefinitely without possibility of making effective progress. A livelock is detected when the state space contains a cycle that leads to no markings outside the cycle. In this case, once the cycle is entered it will repeat forever. A CPN model which terminates properly should be free from livelocks. Therefore, it is important to check that the CPN model of a safety critical interactive device doesn’t contain livelocks, unless intended. There might be a case that the model should have a livelock state, but this is very uncommon. So finding a livelock state is undoubtedly useful and should be a part of any analysis.

It is quite easy to detect livelocks in a small and simple model by simply looking at it. But for real devices, this is not possible. So, there should be some uniform approach that finds out such livelock states.

A convenient way to check the absence of livelocks is to study the automatically generated graph of strongly connected components (SCC graph). Depending on the structure of the generated state space graph, model checking the absence of livelocks takes one of the following two forms [27]:

- If the state space graph and its SCC graph are isomorphic and also there are no self-loops, then the model is free of livelocks;
- If the state space contains self-loops or if there is at least one strongly connected component that consists of more than one node, then we need to examine to see if all terminal components are trivial that is, they consist of a single node and no arcs. A non-trivial component represents a livelock in the model.

The query shown in table V verifies the absence of self-loop terminal nodes.

#### Table V

**Query to verify the absence of self-loop terminal nodes in CPN**

```haskell
fun SelfLoopTerminal n = (OutNodes (n) = [n])
fun InValidTerminal() = PredNodes(EntireGraph, fn n => (SelfLoopTerminal n), NoLimit);```

The code given in table V finds invalid terminals that end in self-loops. The function `SelfLoopTerminal` uses the built-in function `OutNodes` to get a list of output nodes. The argument `PredNodes` specifies a function which maps each node into a Boolean value. The nodes which evaluate to false are ignored; the others take part in further analysis. The value `EntireGraph` denotes the set of all nodes in the state space. The evaluation function `(fn n => (SelfLoopTerminal n))` maps a `SelfLoopTerminal` node into itself. The val `NoLimit` specifies an infinite limit, i.e., that the search continues until the entire
search area has been traversed. If a code returns an empty list, then there are no self loop terminals which means that the CPN model is livelock free.

In our model we get an empty list as in Table VI which means that there are no invalid terminal nodes that end in self loops. Hence the model is livelock free.

### TABLE VI
**DETECTING LIVELOCK IN SIMPLE INFUSION PUMP MODEL**

3) **Checking Values:** It is also important to check the values allowed for a safety-critical interactive system. For example, limits on the volume to be infused or limits on the time of the infusion. The state space method allows us to check if there exist values which violate these limits. For the simple infusion pump, we have a guard $volume < 3$ on the transition $S\_IncreaseVolume$. This means that in the entire state space there should not exist an arc where the value of the variable $volume$ is three or more. We verify this by writing the code written in Table VII to query the state space to see if there is any marking where the value of the variable $volume$ is three.

### TABLE VII
**RETURNS ALL THE ARCS IN THE STATE SPACE GRAPH WHERE THE VALUE OF THE VARIABLE volume IS 3**

```
fun setvolumearcs(volume:NAT): Arc list =
PredAllArcs (fn a=> case ArcToTI a of
| Battery="", infusing="", infusionrate="", timeleft="", timer="", volume=3, volumeleft="")
=> volume = 3 | a => false)
```

Output:
val it = [] : Arc list

We get the empty arc list as an output. This means that there exists no such arc in the entire state space where the value of a variable $volume$ is three. Similarly we can check other values.

4) **Reachability:** Reachability is another important property that needs to be analyzed for safety-critical interactive systems. Reachability means that all system behaviours can be obtained by a user by applying some sequence of commands at some point in their interaction.

Reachability functions in the CPN tool [21] allow checking reachability of one node from another. For example, we want to check whether a user can reach the $setvolume$ state from the $info$ state. For that, we first need to find out the setvolume and info nodes from the state space graph. The functions given in table VIII and IX find the setvolume and info nodes.

We can use the reachability function to check if the nodes given in Table VIII are reachable from nodes given in Table IX. Table X shows the reachability function that checks if we can reach node 99 from node 98, which evaluates to true. We can repeat this process for each of the other pairs.

### TABLE VIII
**RETURNS ALL SETVOLUME NODES IN THE STATE SPACE GRAPH**

```
fun setvolumenodes ()=SearchAllNodes (fn n=> let
fun setvolumearcs(nil)=false |
setvolumearcs(a :: al : Arc list) =
if ArcToTI a = TI.info'I setvolume 1 orelse
ArcToTI a = TI.settime'I setvolume 1 then
true
else
setvolumearcs(al);
in
setvolumearcs(InArcs(n))
end, fn n => n, [], op::);
```

Output:

### TABLE IX
**RETURNS ALL INFO NODES IN THE STATE SPACE GRAPH**

```
fun infonodes ()=SearchAllNodes (fn n=> let
fun infoarcs(nil)=false |
infoarcs(a :: al : Arc list) =
if ArcToTI a = TI.init'I info 1 orelse
ArcToTI a = TI.setvolume'I info 1 then
true
else
infoarcs(al);
in
infoarcs(InArcs(n))
end, fn n => n, [], op::);
```

Output:

### TABLE X
**REACHABILITY FUNCTION TO TEST REACHABILITY FROM info TO setvolume NODE**

```
fun reachable(n1,n2)=
fun reachable'(n1,n2)=
```

Output:
val it=true: bool

### TABLE XI
**REACHABILITY FUNCTION TO FIND THE PATH FROM info TO setvolume**

```
fun reachable'(n1,n2)=
```

Output:
val it=true: bool
If we want to know the path to reach the setvolume node from the info node, the function given in Table XI is evaluated and this tells us the path from the info node to the setvolume node.

Reachability functions provided by the CPN Tool also allow checking total reachability which means the ability to get anywhere from anywhere. The AllReachable function in the CPN tool determines whether all the reachable markings are reachable from each other. This is the case iff there is exactly one strongly connected component [21]. This helps in verifying that all of the behaviours can be accessed by a user and also ensures that a user can return back to the state they came from. Allowing users to return back to the previous states is important as it allows for correcting errors. For the simple infusion pump model, the AllReachable evaluates to false as shown in Table XII.

Of course, our CPN model does not have total reachability as the user cannot return to any other state when in the infusing state.

VII. CONCLUSION AND FUTURE WORK

In this paper we have shown how we can create a CPN model of a safety-critical interactive system that has all the aspects (user interface, interaction and functionality) in a single model using the CPN Tool. We have used the state space analysis method to investigate the behaviour of a system. We have given examples of investigating the behaviour of a simple infusion pump and presented the results. We believe that using this technique can contribute to the safer use of interactive systems. We can use this technique to model and investigate new and existing systems to ensure that these systems behave as intended. We have given an example from the medical domain to illustrate the approach.

In future we would like to apply this technique to other safety critical domains such as the control system of nuclear power plant.

REFERENCES