Addressing Problem Frame Concerns via Coloured Petri Nets and Graphical Animation

Jens Bæk Jørgensen
Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
jbj@daimi.au.dk

ABSTRACT

To address a frame concern in Jackson’s Problem Frames, we must make appropriate descriptions of: (1) the problem domain; (2) the requirements; (3) the specification of the machine. Based on these descriptions, we must give a convincing argument that the given domain properties and the machine specification together entail that the requirements are fulfilled. In this paper, we demonstrate how to address certain frame concerns with the use of the formal modelling language Coloured Petri Nets (CPN). Problem domain description and machine specification are brought together in a CPN model, which is augmented with a graphical animation. The CPN model is executable and we simulate it to address frame concerns. We illustrate the approach on the elevator controller example.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques—Coloured Petri Nets

General Terms

Design, experimentation

Keywords

Problem frames, requirements, specifications, formal models

1. INTRODUCTION

In the terminology of Jackson’s Problem Frames [11], the problem domain is the specific real-world subject matter to be addressed by a new computer system; the machine is the software to be built and the computers that will run the software; the requirements express the desired effect the machine should have in the problem domain; the specification describes the desired behaviour of the machine at the interface with the problem domain, and is to be used by programmers who are about to implement the machine.

Application of the Problem Frames approach demands that we first represent domain properties, requirements, and specifications. Secondly, that we use these representations to address frame concerns, i.e., to make correctness arguments. In some examples in [11], Jackson uses a kind of state machines as a help for this purpose.

The contribution of this paper is to suggest and justify a concrete well-established and well-proven state machine-like language to make the necessary representations and to address certain frame concerns. We propose to use Coloured Petri Nets (CPN) [13, 16], which are a dialect of the broader category of modelling languages of Petri nets [6]. Petri nets are, in general, suitable for describing the behaviour of systems with characteristics like concurrency, resource sharing, and synchronisation. Particular assets of CPN are: (1) CPN provide an extensive state concept, which facilitates the representation of domain properties; (2) CPN are scalable to modelling of large systems; (3) CPN are well supported by computer tools.

We bring together descriptions of domain properties and the machine specification in a CPN model, which is augmented with a graphical animation. Simulation of the model is used to address frame concerns, i.e., to argue that considered requirements are fulfilled. We illustrate the approach on the well-known elevator controller example, which has appeared often in the literature and has been used frequently by Jackson [10, 12, 25].

The motivation behind our proposal to bring together descriptions of domain properties and machine specifications in one model resembles the motivation behind Nuseibeh’s Twin Peaks model [22]. In Twin Peaks, requirements and specifications are developed concurrently and iteratively, and at the same time a clear separation between the description of the problem to be solved and the description of the proposed solution is maintained. Hall et al’s combination of Problem Frames with software architectures [8] is another example of related work.

This paper is structured as follows: Section 2 introduces the problem domain, requirements, and specification for the example. Section 3 presents a CPN model, which brings together descriptions of the previous section. Section 4 describes a graphical animation of the CPN model. In Section 5, we demonstrate how to address frame concerns for the elevator controller using the CPN model and its graphical animation. In Section 6, we reflect more generally on reasons why we believe that CPN are a viable language to be used in conjunction with Problem Frames. Section 7 discusses related work. The conclusions are drawn in Section 8.
2. PROBLEM DOMAIN, REQUIREMENTS, AND SPECIFICATION

The elevator controller we consider must work in a ten floor building in which there are two elevator cages.

The main responsibility of the controller is to control the movement of these cages. Movement is triggered by passengers, who push request buttons. On each floor, there are floor buttons, which can be pushed to call the elevator; a push indicates whether the passenger wants to travel up or down. Inside each cage, there are cage buttons, which can be pushed to request to be carried to a particular floor. In addition to controlling the movement of the cages, the controller is responsible for updating a location indicator inside each cage, which displays the current floor of the cage.

Our rendering of the elevator controller example is an adapted and simplified version of Wieringa’s presentation in [24], which also is the source to the information that is necessary to make the descriptions in this section. Specifically, this means that in this paper, we do not consider how requirements and specifications are found. We just present them — and later focus on their role in addressing problem frame concerns.

2.1 Problem Domain

The problem domain consists of floors, elevator cages, motors, doors, location indicators, buttons, sensors, etc.

The Problem Frames approach specifies that we must carefully make descriptions of these given entities. We will not include such descriptions in this paper, but refer to [24], where Wieringa, among other things, gives a dictionary fixing the meaning of terms in the problem domain (this includes designations in the sense of Problem Frames), a set of entity-relationship diagrams of the problem domain, and a partial context diagram. Moreover, Wieringa describes the assumed behaviour of entities in the problem domain and the desired behaviour of the problem domain to be brought about by the controller; behavioural descriptions are given in the form of a set of statecharts [9].

The following scenario is an example of desired behaviour: Both cages are idle on floor 1. A passenger on floor 4 makes a request for upwards travel by pushing the appropriate floor button. One of the cages starts to move. The cage passes floors 2 and 3, arrives at floor 4, and opens its doors. The passenger enters and the doors close. The passenger pushes a cage button to request to be carried to floor 10; the cage resumes its upwards movement.

2.2 Requirements

There are many requirements that must be ensured by the elevator controller; examples are:

- **Collect passengers:** When a passenger pushes a floor button on floor f, eventually an elevator cage should arrive at floor f and open its doors.
- **Deliver passengers:** When a passenger pushes the cage button for floor f in an elevator cage, eventually the elevator cage should arrive at floor f and open its doors.
- **Show floor:** When a cage arrives at a floor, passengers inside the cage should be informed about the current floor number.

The Collect passengers and Deliver passengers requirements are related to the commanded behaviour problem frame, while the Show floor requirement is related to the information display problem frame.

2.3 Specification

We specify the controller as a set of operations that it must provide; some of the operations are listed and informally described below. All operations but the last one in the list are related to controlling the movement of the cages. The last operation is used to display information in the location indicators.

It is assumed that the controller at any time for each cage will record its current floor, its current direction of movement, and its request list; the latter holds the outstanding requests currently allocated to that particular cage.

- **setdirection:** called when the motor of an idle cage is about to start; determines whether the cage should move up or down, based on the current floor of the cage and its request list.
- **stopere:** called when the cage is in the final approach to a floor, triggered by a sensor; determines whether the cage should stop here. The cage stops if the floor number is in its request list; otherwise it continues. When stopere returns true, it must be ensured that the cage’s doors open upon arrival.
- **turnidle:** called when the cage’s doors are closing; determines whether the cage should turn idle at the current floor. The cage should turn idle if it has no outstanding requests; otherwise, it should continue.
- **servenov:** called when an idle cage with an empty request list receives its first request; determines whether that request can be served immediately. This is possible only if the request comes from the cage’s current floor. In this case, the cage can just open its doors; it is not necessary to start the motor.
- **resetdirection:** called when the doors of a cage with outstanding requests have closed; determines whether the cage should move up or down, based on the current direction of the cage and its request list. If there are any requests in the list that can be served in the current direction, the cage continues. Otherwise, it shifts to moving in the opposite direction.
- **addrequest:** called when a new request is made; adds a request to the request list of a cage.
- **removerequest:** called when a cage opens its doors at a certain floor. The request corresponding to that floor are removed from the cage’s request list. Upon calling removerequest, it must be ensured that the light of the appropriate button is turned off.
- **updatelocationindicators:** called when the cage is in the final approach to a floor; updates the location indicator to show the new current floor.

3. THE CPN MODEL

Based on the descriptions of the previous section, we have built a CPN model, which represents: (1) entities in the problem domain; (2) the specification of the controller; (3) the behaviour of the problem domain as caused by this specification.
(1) is an indicative description, while (2) and (3) are essentially optative descriptions. Thus, the CPN model does gather indicative and optative descriptions. According to Jackson, this should be done very carefully (if at all). However, we hope that we have structured the model and the presentation of it in this section will allow the reader to clearly separate the indicative aspects from the optative aspects.

The model is created and executed with the tool CPN Tools [27], which has a graphical part and includes the programming language Standard ML [19]. Together with the explanation of the model, this section is an informal primer to the CPN language itself, which allows the reader to understand the model in general terms.

The CPN model consists of: (1) declarations of data types, functions, etc.; (2) graphical net structure in the form of three related modules: Do Cage Cycle, Handle Requests, and MoveUpDown. As we will see, the declarations are used as inscriptions in the graphical net structure.

### 3.1 Representation of Problem Domain Entities

Entities in the problem domain are represented via data type declarations. As example, the data type CAGE used to represent the elevator cages consists of 4-tuples of the form (cageid, floor, requestlist, direction); cageid identifies the cage; floor is the number of the floor the cage currently is at, if the cage is stationary, or has last visited, if the cage is moving; requestlist is the cage’s request list represented as a list of floor numbers; direction holds the cage’s current direction of movement: up, down, or no. Other examples are that floors are represented as integers, and a floor button as a pair (floor, direction), where direction is no if the button has not been pushed and otherwise up or down, indicating the passenger’s direction request. The cage buttons in each cage are represented as a pair (cageid, buttonlist), where buttonlist is a list of integers corresponding to the floors for which cage buttons currently have been pushed.

### 3.2 Representation of Controller Specification

Each controller operation listed in Section 2.3 is represented as a Standard ML function with the same name. Examples are shown in Figure 1 and explained below.

```ml
fun setdirection (c,cf,f::rl,cd) = if cf=f then (c,cf,rl,up) else (c,cf,rl,down);
fun stophere (c,cf,rl,cd) = member cf rl;
fun turnidle (c,cf,rl,cd) = (rl=[]);
```

Figure 1: Examples of declarations of functions to represent controller operations.

The `setdirection` function takes a CAGE as argument and returns a CAGE. The return value is identical to the argument, except for the `direction` entry; in the return value, this is set to either `up` or `down`, depending on a comparison of the current floor `cf` and the first floor number in the request list (in Standard ML, a pattern like `head::tail` denotes a list with `head` as first element and `tail` the rest of the list).

The `stophere` function checks if the CAGE argument’s current floor `cf` is equal to the empty list `[]`.

### 3.3 Representation of Problem Domain Behaviour

**Basic cage movement**

The basic movement of cages is described in the Do Cage Cycle module, shown in Figure 2.

![Figure 2: Do Cage Cycle module.](image-url)
shows the current state of the place — that has been done for Idle, where the values of the two CAGE tokens can be seen. For Floor Buttons and Cage Buttons, only the number of tokens can be seen. The remaining places are empty, and no small circles are shown.

All inscriptions in the model starting with init define the initial state of the place, like initCages() for the Idle place; the initX Standard ML functions are not part of the specification of the controller, but a technical means to properly initialise the CPN model.

The events of a CPN model are represented by transitions, drawn as rectangles. Arcs connect transitions with places. The events consists of transitions that remove tokens from input places and add tokens to output places. The expression associated with an arc determines the removed and added tokens; e.g., the expression \((c,cf,f::rl,cd)\) on the arc from the Idle place to the Start Motor transition specify a CAGE token in which the third entry matches the pattern \(f::rl\), i.e., is a non-empty list \((c, cf, f, rl, and cd\) are variables of appropriate data types).

A transition that is ready to remove and add tokens is enabled; it may occur. There are two conditions for enabling: (1) Appropriate tokens are present on the input places — the values in these tokens are bound to the variables appearing in the inscriptions around the transition; (2) A guard — a Boolean expression in square brackets — is true.

In Figure 2, enabling of Start Motor requires that: (1) Idle contains a CAGE token matching the \((c,cf,f::rl,cd)\) pattern, i.e., a CAGE token with a non-empty request list; (2) the guard \([\neg (servenow (c,cf,f::rl,cd))]\) evaluates to true; i.e., that the state represent a situation in the problem domain in which cage \(c\) is not currently at floor \(f\). Start Motor is not enabled in the shown state. However, when other transition occurrences have changed the state of the model, Start Motor can become enabled and can occur. This represents that a cage changes from being idle to be moving: A CAGE token is removed from Idle and a CAGE token is added to Moving; the latter token with its direction entry determined by the call of the setdirection function.

Occurrence of Arrive at Destination represents that a cage arrives at a floor, stops, and opens its doors. This causes updates of tokens representing buttons on the Floor Buttons and Cage Buttons places to represent that the request buttons for the current floor are turned off. Also, the request for the current floor is removed from the request list of the CAGE token by the call of the removerequest function on the arc from Arrive at Destination to Opened.

Occurrence of Close Doors represents that a cage closes its doors (it is assumed that this closing is triggered by some intelligence in the doors themselves, i.e., not caused by the controller; there are no invocation of controller operations around Close Doors). The token representing the cage is put on Closed. The Stop transition is enabled if the CAGE token has an empty request list; this is checked by the turnidle call in the guard. Serve Next Req is enabled if the request list is non-empty. In this way, it is described that the cage either can become idle or resume its movement.

Requests Handling

The handling of requests, i.e., the making and subsequent allocation of requests to cages, is represented in the Handle Requests module, shown in Figure 3.

Each of the places Idle, Floor Buttons and Cage Buttons is conceptually glued with the place with the same name in the Do Cage Cycle module.

Occurrence of the Push Floor Button transition represents that a passenger pushes a floor button, either in direction up or down. In the problem domain, this will cause the floor button to light up, represented in the CPN model by an update of a floor button token on Floor Buttons (it is assumed that turning the light on is done by the button itself, i.e., not caused by the controller; there are no invocation of controller operations around Push Floor Button). That the light of the floor button is eventually turned off again is represented by occurrence of either the Serve or, as we saw, the Arrive at Destination transition in the Do Cage Cycle module (Figure 2).

Occurrence of Push Floor Button has the additional effect that it causes a FLOORREQUEST token to be added to Floor Reqs. Subsequently, this may cause Allocate Floor Req to become enabled; Allocate Floor Req represents assignment of a given floor request to one of the two elevator cages. In the current version of the model, the scheduling policy is very simple: a random idle cage is chosen. We discuss more advanced scheduling policies in Section 6.4.

Occurrence of Push Cage Button represents that a passenger pushes a cage button. In a similar fashion as Push Floor Button, this causes an update of a token on Cage Buttons representing that the button lights up. It also causes a CAGEREQUEST token to appear on Cage Reqs. Occurrence of Allocate Cage Req represents that the request is added to the request list of the cage in which the button is pushed — which, of course, is the only sensible way to allocate a cage request.
Up and Down Movement

The up and down movement of the elevator cages is represented in the Move UpDown module, shown in Figure 4.

The Move place is conceptually glued with the place with the same name in the Do Cage Cycle module.

The Move One Up transition represents movement of an elevator cage one floor up. When it occurs, the floor entry of the CAGE token on Moving is incremented, as can be seen from the expression on the arc from Move One Up to Moving.

The guard of Move One Down consists of three Boolean expressions and should be read as a conjunction. The expressions cd=up and cf<10 represents properties that are inherent to the problem domain: that the cage actually is moving upwards and that it cannot move up if it is at the topmost floor. The expression not(stophere (c,cf,rl,cd)) represents a condition that is enforced by the elevator controller: that the cage only moves if it should not stop — it should stop if there is a request. The Move One Down transition works similarly to Move One Up.

Tokens on the Location Indicators place represent the location indicators for the two elevator cages. The state of Location Indicator is updated each time either Move One Up or Move One Down occurs. The call of the function updatelocationindicators represents the controller’s responsibility of ensuring that each location indicator correctly displays the actual current floor of the elevator cage. In terms of the CPN model, this means that the floor entry of each of the tokens on Location Indicators must be equal to the floor entry of the corresponding CAGE token — which is on one of the places Idle, Moving, Opened, or Closed.

4. GRAPHICAL ANIMATION OF THE CPN MODEL

Figure 5 shows a graphical animation of the CPN model. The middle part of the figure represents the elevator shaft with the two elevator cages. The left part of the figure represents the floor buttons; the right part represents the cage buttons plus the location indicator for each of the two cages.

The graphical animation is interactive. The animation user can make experiments by pushing button icons. The snapshot of Figure 5 represents that cage 1 has stopped at floor 4 and has opened its doors; cage 2 is idle with closed doors at floor 1. Moreover, the animation user has pushed the icon for the downwards button on floor 9. The immediate effect experienced by the user is that the arrow icon has changed colour from white to black, representing the change from light off to light on.

At any time, the animation is consistent with the CPN model. This means that in the situation shown in Figure 5, the CPN model is in the following state: The CAGE token for cage 1 is on the Opened place (Figure 2), the CAGE token for cage 2 is on the Idle place, and the FLOORBUTTON token for floor 9 on the Floor Buttons place (Figures 3 and 2) represents that the down arrow on floor 9 has been pushed.

Whether the state of the CPN model reflects that allocation of the request from floor 9 has happened or not cannot necessarily be seen from a snapshot of the graphical animation. Let’s say that allocation has not happened. In this case, there is a FLOORREQUEST for floor 9 downwards travel on the Floor Req place (Figure 3). Now, the CPN model can emulate that the request from floor 9 is allocated to one of the elevator cages by occurrence of the Allocate Floor Req transition (Figure 3).

Let’s say that the request is allocated to cage 2. In the CPN model, this causes Start Motor (Figure 2) to become enabled. When it has occurred, Move One Up (Figure 4) is enabled. Each time this transition occurs, the animation user will see the icon representing cage 2, and its cage buttons and location indicator panel in the right-hand side of Figure 5, move up one floor in the animation. The animation user will also see an update of the current floor number in the location indicator icon for cage 2. The movement of the cage icon will emulate the passage of all floors below 9, the desired stopping at floor 9, the opening of the cage doors, and the turn off of the down arrow floor button.

In general, the link between the CPN model and the ani-
5. ADDRESSING FRAME CONCERNS FOR THE ELEVATOR CONTROLLER

We will now demonstrate how to address frame concerns for the elevator controller, both directly based on the CPN model and based on the graphical animation.

We consider in detail the Collect passengers requirement (Section 2.2) and argue that the current specification of the elevator controller, together with the given domain properties entail this requirement; we can make similar arguments for the two other considered example requirements, Deliver passengers and Show floor.

5.1 Argument Based on CPN Model

We use simulation to investigate different scenarios and check that in each one, when the model emulates that a passenger pushes floor button \( f \) in the problem domain, eventually the model can be in a state which corresponds to an elevator cage being present at floor \( f \) with its doors open.

As an example, consider a situation in which both elevator cages are idle at floor 1 and there are no outstanding requests. Let’s say that the floor button is pushed at floor 4 for upwards travel. Occurrence of the Push Floor Button transition (Figure 3), with the FLOOR variable \( f \) bound to the value 4 and the DIRECTION variable \( d \) bound to the value up, represents this event; the event is shared between the problem domain and the machine (in the sense of Gunter et al.’s reference model for requirements and specifications [7], the event is a phenomena, which happens in the environment and is visible to the machine).

When the state of the model is in accordance with the assumed problem domain situation, two CAGE tokens are present on Idle. Therefore, Allocate Floor Req (Figure 3) becomes enabled. When it occurs, the value 4 is added to the request list of one of the CAGE tokens on Idle, say the cg(1) token, by the function addrequest. This represents an event, which is internal to the machine (and in Gunter et al.’s sense hidden for the environment).

Now, Start Motor (Figure 2) is enabled; the guard \( \text{not} \left\{ \text{servenow} \left( c,cf,f::rl,cd \right) \right\} \) is true because \( cf=1 \) (the cage’s current floor) and \( f=4 \). The servenow call just checks if \( cf \) is equal to \( f \). When Start Motor occurs, the cg(1) CAGE token is removed from Idle and added to Moving. We can check that the direction of movement is correctly set to up by the expression \( \text{setdirection} \left( c,cf,f::rl,cd \right) \); the setdirection call results in \( cd=\text{up} \) because \( cf=1 \) and \( f=4 \) and thus \( cf < f \) (cf. Figure 1).

Arrive at Destination (Figure 2) is not enabled; the guard \( \text{not} \left\{ \text{stopere} \left( c,cf,rl,cd \right) \right\} \) true because \( cf=1 \) and \( rl=\left[ 4 \right] \) and stopere just checks if \( cf \) is member of \( rl \). However, Move One Up (Figure 4) is enabled and its occurrence causes the floor entry in the CAGE token on Moving to change from 1 to 2. Move One Up is enabled and can occur two additional times, causing the floor entry in the cg(1) CAGE token to be 4.

Arrive at Destination (Figure 2) is now enabled; the guard \( \text{not} \left\{ \text{stopere} \left( c,cf,rl,cd \right) \right\} \) evaluates to true because 4 is member of \( \left[ 4 \right] \). When Arrive at Destination occurs, the cg(1) CAGE token is removed from Moving and added to the Opened place. This corresponds to the problem domain event that elevator cage 1 arrives at floor 4 and opens its doors. Thus, in this case, the Collect passenger requirement can be fulfilled.

We have simulated a number of scenarios, also much more complex scenarios than the one we have explained in detail here. In each case, we have observed that when the model emulates the making of a floor request, we can bring the model in a state that corresponds to the floor request being served.

5.2 Argument Based on Graphical Animation

We have now seen that a frame concern can be addressed by simulation and detailed inspection of the CPN model. But a frame concern can also be addressed via the graphical animation. A user can push floor and cage button icons in the graphical animation (Figure 5). For each push, the user will hopefully experience that the animation eventually shows an elevator cage icon with open doors at the requested floor, and that the location indicator icons are properly updated during the emulation of elevator movement.

If this happens, the graphical animation can be used to validate that the current specification of the controller and the domain properties together can ensure that the requirements are fulfilled, for the considered scenarios. The graphical animation can also be used to discover problems, both simple problems like an elevator cage, which does not stop if it comes to a floor for which it has a request, or more complex problems like the scheduling not being done so that efficient use of the elevator cages is ensured.

However, the graphical animation cannot be used to investigate the causes of and ultimately find solutions to the problems. For debugging, it is necessary to inspect the CPN model itself, including the declaration of the Standard ML functions that represent the controller specification.

6. VIABILITY OF CPN IN ADDRESSING FRAME CONCERNS

A CPN model and its graphical animation can be used to address frame concerns for the elevator controller, as we argued above. More specifically, with the considered requirements, we have addressed two commanded behaviour problems and one information display problem.

We believe that CPN models are more generally applicable to address frame concerns for certain required behaviour, commanded behaviour, and information display problems involving either causal or biddable domains [11]. In this section, we discuss reasons supporting this conjecture.

As structure for our discussion, we will use the five criteria, which Selic puts forwards in [23] as being essential for models to be useful and effective; the criteria seem well applicable to models involving causal or biddable domains. Each subsection below starts with a quotation from Selic’s characterisation of the criteria.

6.1 Abstraction

“A model is always a reduced rendering of the system that it represents. By removing or hiding detail that is irrelevant for a given viewpoint, it lets us understand the essence more easily.”
Certainly, the CPN model of the elevator system is an abstraction. Numerous real-world details are not included in the model, e.g., the obviously irrelevant detail whether it is a male or a female passenger who pushes a button. On the other hand, also many potentially relevant details are not included in the current version of the model. Examples are: (1) Failures: The CPN model could be extended to include strategies for how the elevator controller should handle failures in external entities, e.g., motors, buttons, doors, and sensors; (2) Timing: The correspondence between timing in the real world and timing in the model could be made more precise, e.g., the time it takes to move a cage icon between two floors in the graphical animation could be made approximate with the corresponding real-world time.

6.2 Understandability

“A good model provides a shortcut by reducing the amount of intellectual effort required for understanding”, i.e., a model must be understandable and appealing to the intuition of relevant stakeholders.

The CPN model of the elevator system in itself is only understandable for technically minded stakeholders with proper skills. However, when the CPN model is augmented with a graphical animation, it is possible to communicate with a much broader audience. Many authors have made similar observations and made proposals to use formal models and graphical animations together, see, e.g., [2, 18].

As we saw in Section 5, it is possible to address frame concerns for the elevator controller either directly via the CPN model or via the graphical animation. The former could be possible for software engineers who are about to implement the elevator controller, but probably not for, say, a technical manager at a hotel, who wants to investigate properties of a new elevator system prior to installation. The manager will probably need the graphical animation.

We have previously applied combined, iterative use of CPN models and graphical animations to help users and system developers to reach a common understanding of requirements and specifications; see [14] for a case study in the hospital domain and [15] for a case study in the banking domain.

6.3 Accuracy

“A model must provide a true-to-life representation of the modeled system’s features of interest”, i.e., a model must be accurate in the sense that it is in reliable correspondence with the subject matter being modelled.

CPN have three salient characteristics that facilitate making accurate models: (1) CPN provides an extensive state concept; (2) CPN (as supported by CPN Tools) includes a general-purpose programming language; (3) CPN are scalable, i.e., it is feasible to make CPN models of real-world, industrial-size problems. As an example, it would be trivial to extend our model to convey the elevator system of a high-rise building, say, with 100 floors and 20 elevator cages. No net structure would have to be changed; it would only be a matter of adding more tokens to the CPN model (and make a more flexible graphical animation than we currently have). To be able to properly deal with a problem of this size, however, it would be necessary to carefully choose a sensible scheduling policy for the 20 elevator cages — compared to the very simple policy we use in our 10-floor-2-cages problem.

6.4 Prediction

“You should be able to use a model to correctly predict the modeled system’s interesting but nonobvious properties.” Selic notes that one way to do this is to ensure that a model can be used to make experiments in a trial-and-error fashion. An executable model, such as a CPN model, has this property.

As an example, the CPN model of the elevator system can conveniently be used to experiment with and investigate the consequences of different proposals for scheduling policies: Given a floor request, the controller must decide which of the two elevator cages that should handle the request.

In the current version of the model, as we saw in Section 3.3, any floor request is assigned to a randomly chosen idle cage. If a requirement concerning more efficient and convenient scheduling is considered, we can instantly investigate consequences of proposed changes of the specification via simulation.

As examples, we can change the model to reflect that the request list of a cage can be updated, not only when the cage is idle, but also, e.g., any time when the cage has closed its door after having served a request; in the CPN model, this means when the corresponding CAGE token is on the Closed place (Figure 2). Also, the specification of the controller (Section 2.3) can be extended with an operation smartallocate. Given a request, smartallocate should make a clever assignment of the request to the request list of one of the elevator cages. As straightforward examples, smartallocate could enforce that a floor request is assigned to the nearest cage or to the cage with the shortest request list; smartallocate could take advantage of the direction of floor button pushes (the current specification of the controller just registers that there is a push from a certain floor; it does not distinguish between upwards and downwards pushes). We can include considered proposals in the CPN model by changing some data types and add a guard like \( \text{c = smartallocate (f,d) \ldots} \) to the Allocate Floor Req transition (Figure 3).

6.5 Inexpensive Cost

“A model must be inexpensive — that is, it must be significantly cheaper to construct and analyze than the modeled system.” Selic focuses mostly on models of software, while Problem Frames emphasise also making descriptions or models of the problem domain.

Therefore, with respect to our approach and example, the relevant cost-benefit question to ask is not whether the CPN model of the elevator system is much cheaper than the real system — it obviously is — but whether it is worthwhile to make such a model in the development of the elevator
controller.

In general, the cost of creating and using CPN models is a potential problem, which CPN share with many other formal methods. Formal methods are often seen, not the least by the software industry, as having a negative effect on productivity. Although many good reasons to use formal methods can be given, e.g., that they facilitate finding critical design errors and flaws early rather than late, Neill and Laplante’s survey [20] of the state-of-the-practice of requirements engineering in the software industry finds that formal models are only rarely used.

Whether use of CPN to address frame concerns is cost-effective is an open question; it will require substantial and difficult future work to find reliable indications.

7. RELATED WORK

In this section, we compare the approach presented in this paper with related work.

A number of authors are working on combining Problem Frames with UML. Choppy and Reggio have demonstrated how to address frame concerns for the elevator controller by the use of various UML diagrams [3]. There are several advantages of using UML, being the de facto modelling language of the software industry. On the other hand, use of UML in general also comes with well-known problems like overwhelming size and incomplete, unclear semantics.

We can make a specific comparison of the use of CPN with use of UML for the elevator controller, based on both Choppy and Reggio’s specification in [3], and on Wieringa’s specification in [24]. Both these specifications use statecharts to describe the desired life cycle of one single elevator cage. In comparison, the CPN model explicitly describes the two elevator cages in the same model. Therefore, we believe that the CPN model is a better basis for addressing one of the main issues to be dealt with by the elevator controller, namely scheduling. As we argued in Section 6.4, the CPN model is useful for making experiments with various proposals for scheduling policies.

Lavazza and Del Bianco have given another example of combining Problem Frames and UML [17]. Here, the very notation used by Jackson in [11] is criticised for not being easy to use and for not being tool supported. Our suggestion to use graphical animation and CPN Tools including the Standard ML programming language is a possible step towards addressing this critique.

Nelson et al have presented a case study where frame concerns are addressed via formal analysis [21]. Problem domain, machine, and requirements are described in D. Jackson et al’s formal modelling language Alloy [26]. CPN are a formal modelling language as well and CPN have a variety of formal verification techniques and tools. Therefore, the CPN model of the elevator system is amenable to formal verification. Requirements can be formalised in terms of temporal logics formulae on states and transition occurrence sequences.

State space analysis [13] (model checking) may enable a more solid address of the frame concerns than we did in Section 5. Here, we merely argued that for each considered request, the CPN model can be brought in a state corresponding to the request being served; we did not argue that this does always happen. Moreover, we may be able to analyse more advanced behavioural properties than is possible with simulation, such as fairness properties, e.g., the possibility of requests being starved. However, state space analysis is hampered by the state explosion problem and to analyse our elevator controller CPN model, advanced techniques for state space reduction, condensation, and exploration (see, e.g., [4]) are needed. Unfortunately, such techniques are not yet properly supported by CPN Tools.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed CPN as a language to be used to address certain Problem Frame concerns. We illustrated the proposal on the elevator controller example and reflected more generally on reasons for CPN’s viability.

A limitation of our work is that we have only addressed frame concerns in a flat fashion, without decomposition. In general, for successful use of Problem Frames in real software development projects, it is necessary to be able to decompose larger problems into smaller problems and to combine the addressing of each of the frame concerns for the smaller problems into the addressing of the frame concern for the larger problem. How to do deal with this in a CPN-based approach is an issue for our future research; we hope that CPN’s module concept can be a help.

Problem Frames are an approach for structuring, analysing, and describing software development problems, more than it is an approach for devising solutions. Problem Frames do not address how to bridge the gap between a specification and the implementation of a machine. Neither do we with our proposal to use CPN to address frame concerns. The CPN model merely specifies the elevator controller in terms of a set of Standard ML functions — and separately, in the graphical net structure, describes the effect that the specification has in the problem domain. There is a gap between the specification and the implementation of the controller, both because (1) the CPN model does not cover all relevant issues that the implementation must eventually deal with, e.g., the logics in handling of failures of external entities like motors, doors, and sensors; but also because (2) the CPN model does not deal with the implementation platform for the controller.

Issue (1) above can be dealt with by more work on the CPN model. Issue (2), however, is difficult. In the terminology of Model Driven Architecture (MDA) [28] and Model Driven Development (MDD) [23], the CPN model of the elevator system is a platform-independent model (PIM). How to take the step from a CPN-based PIM to a platform-specific model (PSM), e.g., to the real implementation of the elevator controller, is a subject for ongoing and future research [1].

9. REFERENCES


