Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller*

João M. Fernandes[†], Jens Bæk Jørgensen, Simon Tjell
Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
{jmf,jbj,tjell}@daimi.au.dk

Abstract

This paper presents a model-based approach to requirements engineering for reactive systems; we use an elevator controller as case study. We identify and justify two key properties that a model which we construct must have, namely: (1) controller-and-environment-partitioned, which means constituting a description of both the controller and the environment, and distinguishing between these two domains and between desired and assumed behaviour; (2) use case-based, which means constructed on the basis of a given use case diagram and reproducing the behaviour described in accompanying scenario descriptions. For the case study, we build an executable model in the formal modelling language Coloured Petri Nets. We demonstrate how this model is useful for requirements engineering, since it provides a solid basis for addressing behavioural issues early in the development process, for example regarding concurrent execution of use cases and handling of failures.

1 Introduction

A reactive system is "a system that is able to create desired effects in its environment by enabling, enforcing, or preventing events in the environment" [14]. This characterisation implies that in requirements engineering for reactive systems it is necessary to describe not only the system itself, but also the environment in which the system must operate [8]. In this paper, we suggest a model-based approach to requirements engineering for reactive systems; we illustrate our proposal in the development of an elevator controller, which is a standard example in the literature.

In our approach, we must construct an executable model that is *controller-and-environment-partitioned*. This key property means that it is a description of (1) the desired behaviour of the controller itself; (2) the desired behaviour of the composite system that is made up of the controller, plus relevant external entities in its environment; and (3) the assumed behaviour of these external entities. Additionally, the description must clearly distinguish between the controller and the environment and also between desired and assumed behaviour. For example, desired behaviour is that an elevator car stops when it comes to a floor for which there is a request; assumed behaviour is that the motor starts and sends the car downwards when it receives a goDown signal. The reason that we emphasise the distinction between desired and assumed behaviour is that developers have freedom to make design choices regarding the former, while regarding the latter they "just" have to accept and understand what is given and act accordingly.

The second key property of the model is that it must be *use case-based*, which means that it is constructed from a given use case diagram and can reproduce the behaviour described in accompanying descriptions of scenarios for the use cases. The reason that we enforce this property is that use cases are a convenient and widely-used technique and we would like our approach to be useful in projects that apply use cases for requirements engineering, because this may increase the possibility of its industrial adaptation. Our approach expands, refines, and supplements use case descriptions through the creation of a model, which can be seen as an executable version of a given use case diagram.

The model-based approach that we suggest deviates from what is often done. If we follow, say, RUP [12], after creating the use case diagram and the accompanying scenario descriptions we would most likely soon create a class diagram of the software system to be made. Instead, here we focus on the entire elevator system by creating a behavioural model, prior to more technical software design.

In this paper, we demonstrate how to create a model in the *Coloured Petri Nets* (CPN) language [9] that has the two properties which we pursue. In this way, we combine UML use case diagrams and sequence diagrams with

[*] Research supported by FCT, under project SOFTAS (contract POSC/EIA/60189/2004).

[†] On a sabbatical leave from University of Minho, partially supported by FCT, under grant SFRH/BSAB/607/2006.

1530-1362/07 $25.00 © 2007 IEEE
DOI 10.1109/ASPEC.2007.15

IEEE computer society

CPN. Hence, we contribute to a more general effort on combining UML notations and Petri nets (see, e.g., [3, 6]), which we believe can be useful in software engineering and therefore deserves more attention than it has had. Using the CPN model in requirements engineering for an elevator controller allows some behavioural issues to be analysed and dealt with early in the development process. An example is concurrent execution of use cases and related issues, like resource access and synchronisation. Another example is failure handling; many failures can happen in the environment and must be detected and handled by the controller.

The paper is structured as follows. Sect. 2 introduces the use case diagram and the scenario descriptions for the elevator controller case study. A brief introduction to the CPN modelling language is given in sect. 3 to provide the reader previously unacquainted with CPN the necessary background to understand this paper. The CPN model of the case study is presented in sect. 4, and its use to address behavioural issues is discussed in sect. 5. We cover related work in sect. 6 and draw some conclusions in sect. 7.

2 Use Cases and Scenarios for the Case Study

We consider a simplified version of an elevator system with two elevator cars in a six-floor building. The main responsibility of the elevator controller is to control the movements of the cars, which are triggered by passengers pushing buttons. On each floor, there are hall buttons that can be pushed to call the elevator; a push indicates whether the passenger wants to travel up or down. Inside each car, there are floor buttons, which can be pushed to request the car movement to a particular floor, and there is one button to force the car door to open. The controller is also responsible for updating a floor indicator inside each car that displays the current floor of the car. Similarly, a direction indicator must be updated. Fig. 1 shows the use case diagram for the elevator controller. The use cases are briefly described below.

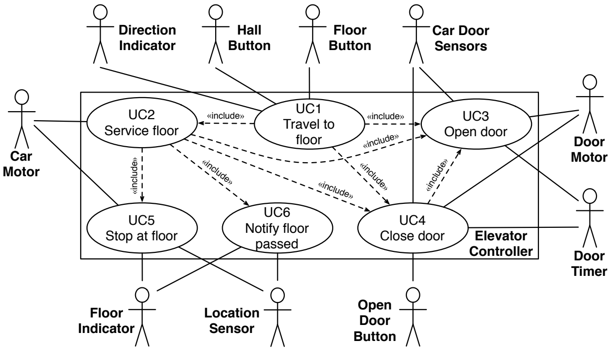

Figure 1. Use case diagram for the elevator controller.

- **UC1 Travel to floor** calls an elevator car to the floor that has been requested, and after the elevator car arrives travels to a given destination floor.
- **UC2 Service floor** moves the elevator car from one origin floor to a destination floor.
- **UC3 Open door** opens the car door.
- **UC4 Close door** closes the car door. If the door is blocked or the open door button is pushed while the door is being closed, the door is opened and will close again after a timer expires.
- **UC5 Stop at floor** stops an elevator car at a given floor.
- **UC6 Notify floor passed** informs the passengers inside the elevator that a given floor is about to be passed.

As usual, the use case diagram identifies and names the use cases that the elevator controller must support, and shows the external actors participating in the use cases. The actors in the use case diagram are the external entities that the controller interacts directly with. These entities are given, and we cannot change them or affect them in our development project, but we must know how they behave. This conception of actors in a use case diagram may deviate from more common conventions. Many use case diagrams for the elevator controller would include a "passenger" actor. We do not do this, because the elevator controller does not interact directly with passengers, but merely with buttons and sensors operated and affected by passengers.

To describe the individual use cases in detail, their textual descriptions are supplemented with sequence diagrams that specify some behavioural scenarios accommodated by the use cases. These scenarios describe desired behaviour of the elevator system, consisting of the controller plus its external actors. As an example, the description of the main scenario for UC1 is presented next, including references to the sequence diagram that is depicted in fig. 2:

1. The passenger at floor f_o requests an elevator to travel in a given direction (specified in the sequence diagram by message carRequest from Hall Button to Elevator Controller);
2. Internally, the Elevator Controller selects a car and assigns it to the request (messages selectElevatorCar and assignRequest);
3. The passenger is notified that the request has been assigned (message notifyCarSelected from Elevator Controller to Hall Button);
4. The car is sent to floor f_o (1st *ref* to UC2);
5. The passenger is notified that the car has arrived at floor f_o (message notifyCarArrived from Elevator Controller to Hall Button);
6. The car door is opened (1st *ref* to UC3);
7. The passenger is notified of the direction the car will take (message light from Elevator Controller to Direction Indicator);

8. After entering the elevator, the passenger selects destination floor f_d (message destinationFloorSelected from Floor Button to Elevator Controller);
9. Internally, the Elevator Controller registers the request (message recordRequest);
10. The passenger is notified that the request has been registered (message notifyRequestFloorRecorded from Elevator Controller to Floor Button);
11. The car door is closed (*ref* to UC4);
12. The car is sent to destination floor f_d (2nd *ref* to UC2);
13. The passenger is notified that the request to travel to floor f_d has been served (message notifyRequestFloorServed from Elevator Controller to Floor Button);
14. The car door is opened and the passenger exits the elevator (2nd *ref* to UC3).

Similar textual descriptions and sequence diagrams exist for the other use cases. There are some dependency relationships among the use cases (for example, UC1 needs the functionalities provided by UC2, UC3, and UC4). They are specified in the use case diagram by *include* relations and in the sequence diagrams by *ref* operators.

3 Brief Introduction to Coloured Petri Nets

This section gives a brief and informal introduction to the CPN language. For a more complete and formal treatment, please refer to [9, 10]. CPN is a well-proven formal modelling language, suitable for describing the behaviour of systems with characteristics like concurrency, resource sharing, and synchronisation. The CPN language is supported by *CPN Tools* (www.daimi.au.dk/CPNTools) which is licensed to more than 4,000 industrial and academic users. CPN Tools facilitates construction, editing, execution, and analysis of CPN models. The CPN language provides an explicit description of both states and actions, and gives a modelling convenience corresponding to a high-level programming language with support for data types, modules, and hierarchical decomposition.

A CPN model is a graphical structure, composed of *places*, *transitions*, *arcs*, *tokens*, and *inscriptions*, supplemented with declarations of *data types*, *variables*, and *functions*. An example of a CPN model, representing the behaviour of the door timers used in the elevator system is shown in fig. 3. The tokens may have complex data values (*colours*). The use of functions and expressions to manipulate data values allows the complexity of a model to be split between graphics, declarations, and inscriptions.

Places are drawn as ellipses and hold multi-sets (bags) of *tokens*. A place models a local state, given by its tokens. The global state of a model is the union of all local states.

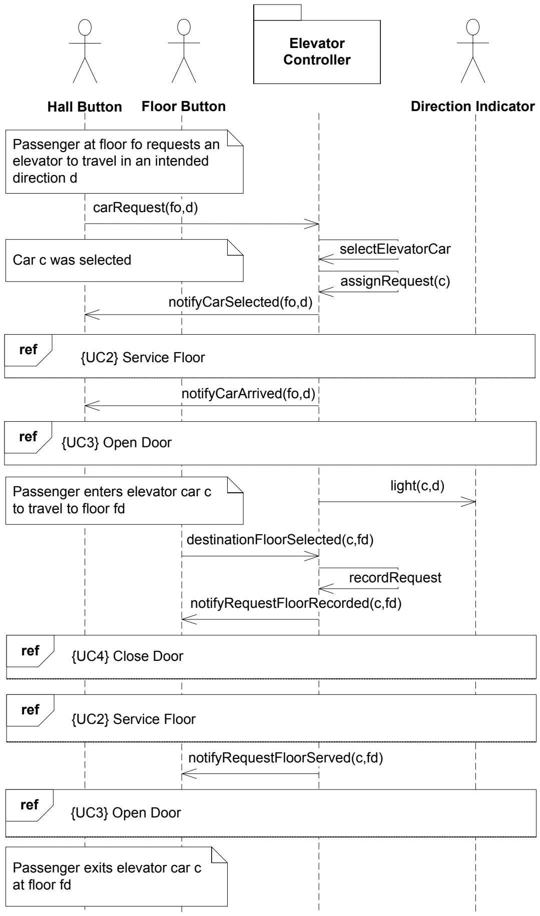

Figure 2. Sequence diagram for the main scenario for UC1.

Each place has an associated type, indicated by an inscription near the place, that specifies the kind of tokens that it may contain. For example, the places Stopped and Running in fig. 3 can both contain tokens of the type CarID_t. The place Stopped has two tokens, indicated by the circled "2", and the place Running has no tokens. The two tokens in the place Stopped represent two door timers, one for each elevator car. One token has the value 1, and the other has the value 2 (indicated by the expression 1'1++1'2 in the box near the place). These values are used to distinguish the timers of the two cars. This shows an essential feature of the CPN language: the pattern of behaviour of the door timers is expressed only once by the structure of their CPN module, and multiple timers operating concurrently are modelled by tokens with individual IDs.

Transitions model behaviour and are drawn as boxes. A transition is connected to *input places* and *output places* by *arcs*. The state of the model is changed when transitions *fire*, by moving tokens from input to output places and/or by changing the values of the tokens. The door timer has two different states represented by the two white places in fig. 3: it is either running or stopped. The black places are used to model the interface between the door timer and the controller (more details in sect. 4.1). Two events can occur in the door timer and each of these is represented by a transition: the timer either expires or is started.

The Start transition has the places Stopped and Ctrl.-Controlled Shared Events as input places and the place Running as output place. When the transition fires (representing the event that the timer is started), one token is removed from each of its input places and a new token is added to its output place. A transition is said to be *enabled* (i.e., ready to fire), when it is possible to consume a collection of tokens from its input places that complies with the restrictions expressed by the inscriptions on the arcs connecting these places to the transition.

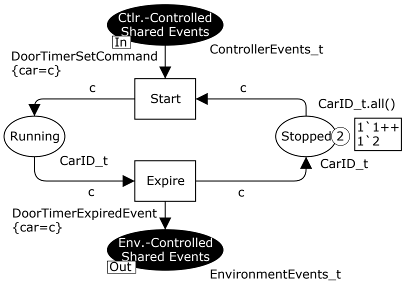
Ctlr.-Controlled Shared Events
In
DoorTimerSetCommand {car=c}
ControllerEvents_t
c
Start
c
CarID_t.all()
Running
Stopped
2
1`1++ 1`2
CarID_t
Expire
CarID_t
c
c
DoorTimerExpiredEvent {car=c}
Env.-Controlled Shared Events
Out
EnvironmentEvents_t

Figure 3. The Door Timer **module.**

The Start transition is enabled when: (1) there is a token in the Stopped place, representing that a timer is stopped, and (2) there is a token in the Ctrl.-Controlled Shared Events place, representing that a command has been sent to the controller to start the timer. The variable c is used in both input arcs of the Start transition to relate the token representing the occurrence of an event (the command) and the token representing the current state of the timer.

Substitution transitions (of which there are none in fig. 3) constitute the basic mechanism for arranging a CPN model in a hierarchical structure. A substitution transition, graphically represented by a doubled-edged box (see, e.g., fig. 5) is a transition that stands for a whole module of the CPN structure. A substitution transition in a super-module is connected to its sub-module via places on the two modules, which are conceptually glued together. Regarding structuring of models, substitution transitions serve the same role in CPN models as superstates do in statecharts [7].

4 CPN Model for the Case Study

In this section, we present the CPN model that has been created for the elevator controller case study, based on the artefacts presented in sect. 2. The CPN model is structured in a hierarchy of which the topmost modules are shown in fig. 4 (these modules are sufficient for this paper, and we do not have space to present the other modules of the CPN model). The boxes in the figure represent modules while the connecting lines represent relationships in the hierarchy. For example, the Top module contains two substitution transitions: one bound to the Controller module and one bound to the Environment module.

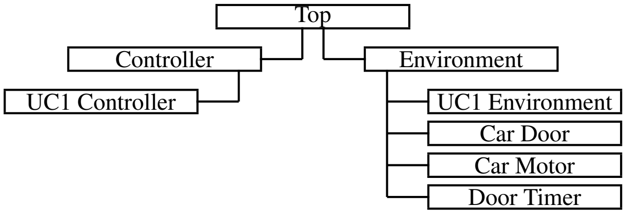

Figure 4. The hierarchy of CPN modules.

4.1 The Top Module

The topmost module of the CPN model (fig. 5) is structured to describe the controller and the environment. In this way, the CPN model is constructed to ensure the first key property we pursue; it is controller-and-environment-partitioned (the distinction between desired and assumed behaviour in the environment is dealt with on the lower levels in the hierarchy). Each of the two domains is represented at the top level by a substitution transition.

The two domains communicate through an interface formed by a collection of shared phenomena [8]. A *shared phenomenon* is a state or an event that is observable by both domains while being controlled by only one domain. In contrast, a *private phenomenon* is only observable within the controlling domain (not to be confused with the controller domain). In our case study, an example of a shared phenomenon is the event of a passenger pushing a request button. It occurs in the environment and is observable by the controller. The controller records all pending requests in an internal data structure, and this is a private phenomenon of the controller. In CPN models, places can not only be used for holding information about states, but also for exchanging information about the occurrence of events. Thus, a place can be seen as a communication channel between two modules. In the model, we enforce a distinction, firstly, between shared events and shared states and, secondly, between phenomena controlled by either the controller or the environment, which results in four places as shown in fig. 5.

The place Ctlr.-Controlled Shared States holds tokens with values representing shared states that are controlled by the elevator controller. Examples of these states are the states of the signals (interpreted as electric voltages) used to control the car motor, the direction indicator lights, and the door motor. The place Ctlr.-Controlled Shared Events holds tokens that represent the information indicating that particular events caused by the controller occurred. An example of such an event is the start of the door timer. The place Env.-Controlled Shared States holds tokens that represent shared states controlled by the environment. Examples of such states are the readings from door sensors, and the state of the car door timer. The place Env.-Controlled Shared Events holds tokens that represent the information indicating that particular events caused by the environment occurred. Examples of such events are when a button is pushed, a timer expires, and a car is sensed near a given floor by a location sensor.

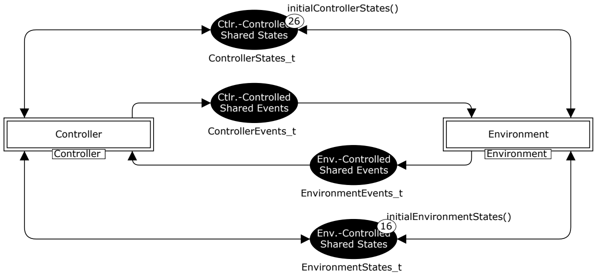
initialControllerStates()
26
Ctlr.-Controlled Shared States
ControllerStates_t
Ctlr.-Controlled Shared Events
ControllerEvents_t
Controller
Controller
Environment
Environment
Env.-Controlled Shared Events
EnvironmentEvents_t
initialEnvironmentStates()
16
Env.-Controlled Shared States
EnvironmentStates_t

Figure 5. The Top **module.**

Since the shared phenomena places only contain tokens that represent shared phenomena, the structure defined in the Top module helps to ensure that the CPN model possesses the environment-and-controller-partitioned property.

4.2 The Controller and Environment Modules

Fig. 6 depicts the Controller and the Environment modules that, as can be seen in fig. 4, are sub-modules of the Top module. Both modules contain a substitution transition used in the modelling of UC1. The substitution transition UC1 Travel to Floor (Controller) represents all the actions initiated by the controller, while the substitution transition UC1 Travel to Floor (Environment) represents all the actions initiated by the environment as part of UC1. Both figures include black places holding representations of shared phenomena; they are conceptually glued together with the places with the same names in the Top module. Thus, the two modules for UC1 communicate through the interface formed by the black places exclusively. This is an example of how the controller-and-environment-partitioned property guides and restricts the way a use case is modelled.

In the Environment module, three external entities (Car Door, Car Motor, and Door Timer) are represented by the correspondingly named substitution transitions. The Door Timer module is shown in fig. 3. Only external entities that contain both private and shared phenomena are modelled by substitution transitions; entities without private phenomena are represented by tokens in the black places (e.g., Location Sensor and Floor Indicator). In fig. 3, the substitution transitions representing the external entities model assumed behaviour, whereas UC1 Travel to Floor (Environment) models desired behaviour. In the Controller module, there is no assumed behaviour, since we are developing the controller and can give it any behaviour that we desire.

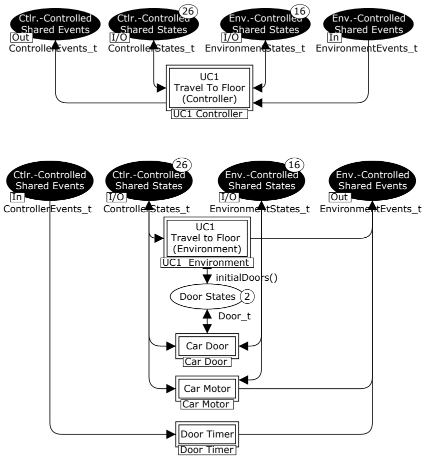
Ctlr.-Controlled Shared Events
Out
ControllerEvents_t
Ctlr.-Controlled Shared States
26
I/O
ControllerStates_t
Env.-Controlled Shared States
16
I/O
EnvironmentStates_t
Env.-Controlled Shared Events
In
EnvironmentEvents_t
UC1 Travel To Floor (Controller)
UC1 Controller
Ctlr.-Controlled Shared Events
In
ControllerEvents_t
Ctlr.-Controlled Shared States
26
I/O
ControllerStates_t
Env.-Controlled Shared States
16
I/O
EnvironmentStates_t
Env.-Controlled Shared Events
Out
EnvironmentEvents_t
UC1 Travel to Floor (Environment)
UC1 Environment
initialDoors()
Door States
2
Door_t
Car Door
Car Door
Car Motor
Car Motor
Door Timer
Door Timer

Figure 6. The Controller **(top) and** Environment **(bottom) modules.**

This completes our justification that the CPN model has the controller-and-environment-partitioned property. We have described both the controller and the environment and we have distinguished between them (on the Top) and between desired and assumed behaviour.

4.3 The Modules for UC1

Fig. 7 shows how the sequence diagram in fig. 2 has been translated into two CPN modules that specify the behaviour of UC1: one for the environment and one for the controller. Messages in the sequence diagram are represented as substitution transitions in the CPN modules. An example from the top of the sequence diagram is the carRequest message, which gives rise to the substitution transition by the name Passenger Requests An Elevator in the Environment module for UC1. This message is followed by two internal messages generated in the controller: selectElevatorCar and assignRequest. These messages and the notifyCarSelected message are represented by the Select Car, Assign Request, Notify Selected substitution transition in the UC1 Controller module. The subsequent messages in the sequence diagram are reflected in the CPN model in a similar fashion.

The two CPN modules for UC1 communicate through the black places holding shared phenomena. Thus the progress of the environment part of UC1 is observable to the controller part through the places Env.-Controlled Shared Events and Env.-Controlled Shared States, while the progress of the controller part of UC1 is observable to the environment part through Ctrl.-Controlled Shared Events and Ctrl.-Controlled Shared States.

The other five use cases in fig. 1 are also reflected in the CPN model, in a similar fashion. Thus, the CPN model is use case-based. In summary, we have achieved a CPN model that has the two properties that we have pursued.

5 Addressing Behavioural Issues

This section discusses how the CPN model can be used to address a number of behavioural issues of the elevator controller, related to concurrent execution of use cases and handling of failures and unexpected uses.

5.1 Concurrent Execution of Use Cases

The CPN model has a parameter NumberOfPassengers that specifies the number of passengers, initially interested in using the elevators. This value determines the number of tokens that are initially put in place Passengers Ready To Request Elevator (fig. 7). If NumberOfPassengers is greater than one, the model reflects the possibility of UC1 (and the included use cases) being executed in multiple concurrent instances. One instance of the use case (i.e., one scenario) is initiated every time an event representing a passenger pushing one of the hall buttons occurs in the CPN model.

If, for instance, NumberOfPassengers is set to ten, the CPN model can mimic up to ten passengers requesting the elevator. This means that a maximum of ten tokens can be accumulated in the Env.-Controlled Shared Events place, after an equal number of firings of an internal transition in the sub-module bound to the substitution transition Passenger Requests An Elevator. Each token represents the completion of the event of a passenger pushing a hall button, and each event initiates an individual instance of UC1.

The tokens generated in the UC1 Environment module that represent hall button push events are put in place Env.-Controlled Shared Events. Each event must be assigned to a given car and this decision, modelled by the substitution transition Select Car, Assign Request, Notify Selected, is taken by the controller, based on internal information (e.g., reserved cars) and external information (e.g., location of cars). The place Assigned Requests holds all pending requests to be served. Whenever a given car is available (represented by a token in place Available Cars), one of the pending requests is picked according to the scheduling

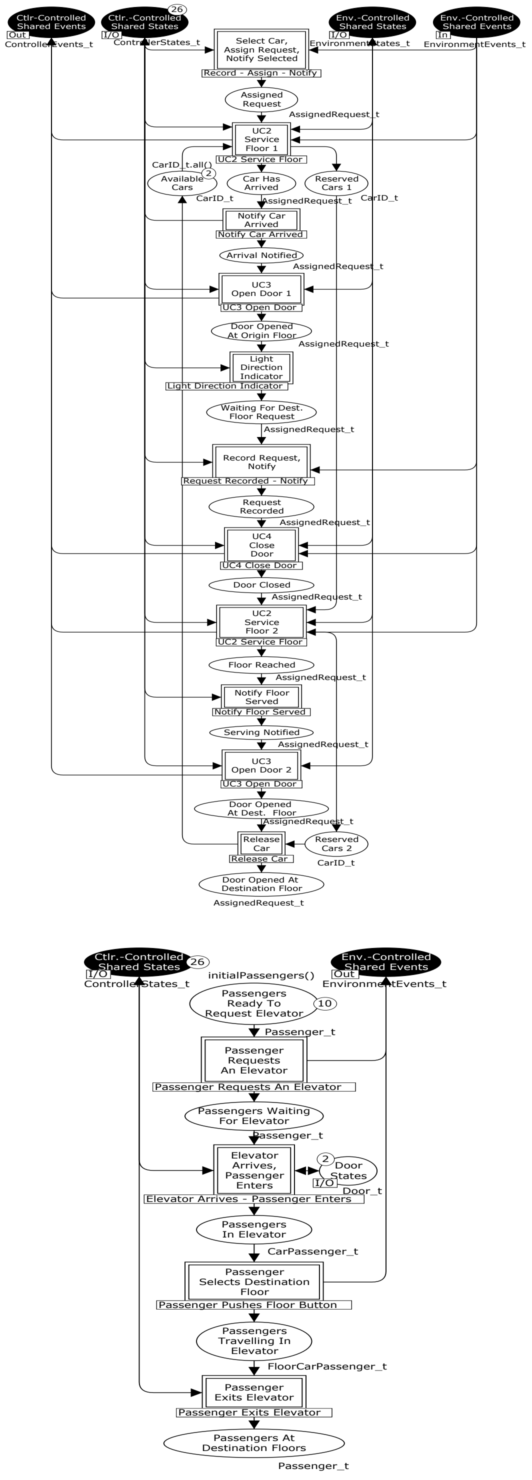

Figure 7. The UC1 Controller **(top) and** UC1 Environment **(bottom) modules.**

strategy, and the assigned car is sent to the floor where the request happened.

The controller must respond to all these events, to actually transport passengers. The CPN model permits the developer to experiment with how these events should be handled by the controller, e.g., to investigate whether the selected scheduling algorithm obtains an acceptable usage of the elevator cars, which are the shared resources in the elevator that the passengers compete for.

5.2 Failures and Unexpected Uses

The division between controller and environment in the CPN model supports simulation of situations where the latter does not behave as expected. An example is a failure of the door timer: when the timer is started, it is assumed to expire after a fixed period of time. This is signalled to the controller, which reacts by closing the door. The model of the environment can easily be modified to include a situation where the timer does not expire, by adding a transition to the module in fig. 3 that moves a token from the Running place to the Expired place without producing an event.

The CPN model can also be used to investigate unexpected uses. An example is when a passenger pushes a hall button. When the car arrives, the passenger is assumed to enter the elevator and select a destination floor. The UC1 Environment module (fig. 7) can be modified to include a scenario where the passenger leaves the location without waiting for the elevator. This can be done by adding a transition that consumes a token from the Passengers Waiting For Elevator place.

In both examples, the reaction of the controller can be observed by executing the model, which can be iteratively adapted to describe how to handle the situation. Any controller must deal with issues similar to those we have discussed here. It is beneficial to address them early, and the CPN model provides a means to facilitate this.

6 Related Work

In [1], it is shown how to make correctness arguments in the context of Jackson's problem frame [8] for an elevator controller by the use of various UML models, which are also controller-and-environment-partitioned. There are several advantages of using UML, the standard modelling language of the software industry. However, UML is not a perfect or universal modeling language. From a technical perspective, UML can sometimes benefit from being supplemented by other modelling languages, e.g., CPN as we have used in the considered case study.

We can compare the use of CPN with the use of UML for the elevator controller, described in [1] and in [14], which employ statecharts to model the desired life cycle of one single elevator car. In comparison, our CPN model explicitly describes multiple elevator cars. Thus, our CPN model is a more accurate description of the real world that the elevator controller controls, with the concurrent behaviours that the real world exhibits.

In [4], it is suggested to utilise use case diagrams and scenarios to obtain one hierarchical CPN model of the behaviour of an interactive system. The hierarchy of the CPN model mimics the one of the use case diagram. The usage of the colours in the nets preserves the independence of several scenarios after their integration in the CPN model. This permits modeling of concurrency between use case, scenarios and copies of the same scenario. However, this approach only tackles the controller perspective, and does not explicitly describe the environment.

The approach in [2] suggests use cases to be described by tables, to ease the communication between the analyst and the domain expert. Later, through some mapping rules, Petri nets are built from the tables to formalise the requirements. The approach is used for producing object-oriented requirements specifications, based on structural models and focuses on deriving intra-object behavioural models. Again, the environment is not explicitly modelled.

7 Conclusions

We have suggested through a case study a model-based approach to requirements engineering for reactive systems. We have also presented a CPN model that has two key properties, namely being controller-and-environment-partitioned and use case-based. Further research is needed to investigate how the approach can be generalised.

As discussed in the previous section, the CPN model for the elevator controller case study allows various issues regarding behaviour to be addressed. In addition to the discussion there, it should be noted that it is possible to apply the standard analysis techniques for CPN (e.g., model checking techniques) documented in the literature for further analysing behavioural properties, and also to use associated tools for edition, simulation, and animation.

Another advantage provided by CPN is that it allows models to be constructed in a parameterisable way. In our case study, we have considered a specific elevator system with two cars in a building with six floors. If we wanted to deal with other elevator systems, this would be straightforward. Adapting the elevator controller for distinct contexts (e.g., different number of passengers, cars, or floors) requires no changes to the structure of the CPN model, but only some specific parameters to be modified.

The created CPN model does not synthesise the complete behaviour of the controller, since our focus is on earlier stages of the software development process. However, we deem that the CPN model can be useful in later development stages as well, because of its structure that separates the description of the controller from the description of the environment. The CPN model may be used within a model-driven development in the sense that parts of platform-specific code for the actual controller might be generated automatically from the part of the CPN model that represents the controller. In the context of model-based testing, it may be possible to use the CPN model for automatic generation of a large collection of test cases; the clear interface between the descriptions of the controller and the environment supports black-box testing.

It may very well be possible to use other modelling languages than CPN to create models with the two desired properties, e.g., statecharts, UML state machines, or UML activity diagrams. Indeed, many of the points that we make about CPN in this paper may also be valid for UML 2.0 activity diagrams. Compared to UML 1.x, UML 2.0 has introduced several modifications to activity diagrams, which are no longer a special type of state machines (ActivityGraph used to be a subclass of StateMachine in the Metamodel) and whose meaning is now explained with concepts borrowed from Petri nets.

If we aim to obtain a controller-and-environment-partitioned model, a modelling language that supports description of concurrency issues seems essential. If we only wanted to describe the controller itself, it may not have been necessary to describe concurrency issues; the controller may well be single-threaded. Since we also want to describe the environment, concurrent execution must be described, e.g., the two elevator cars are travelling up and down concurrently, and the motor is running while the sensors are being activated. The need to describe concurrency puts demands on the chosen language, and CPN is particularly well-suited to handle these issues.

The use of CPN has a number of advantages, but also some problems. As we argue in [11], CPN satisfies four of the five criteria, which are put forward as being essential for good modelling languages in [13]. CPN models (1) are *abstract*, (2) can be made *understandable* (in particular when the CPN model itself is hidden behind a graphical animation, e.g., [11]), (3) can be made *accurate*, and (4) can be used for *prediction*. However, it is not known whether CPN models do or do not satisfy the fifth criteria, that models must be *inexpensive*. The cost-effectiveness of using CPN has not been established, which is an issue that the CPN language seems to share with many formal methods.

We plan to mature and generalise the suggested approach. Currently, we consider exactly one sequence diagram for each use case. We need to study how to deal with more than one sequence diagram for a use case, as we do in [5]. We expect also to apply the approach in industrial contexts, to evaluate its usefulness in helping developers of reactive systems in their requirement engineering activities. This requires a careful analysis of the impact of introducing the approach on the software process and on the tools in use. Defining some guidelines and providing some automatic means to obtain the CPN model from the use cases diagrams and the sequence diagrams are important issues to ensure that the approach can be applied in a sensible way.

References

[1] C. Choppy and G. Reggio. A UML-Based Method for the Commanded Behaviour Frame. In *1st Int. Workshop on Advances and Applications of Problem Frames (IWAAPF '04), at ICSE 2004*, pages 27–34. IEE, 2004.

[2] B. Dano, H. Briand, and F. Barbier. An Approach Based on the Concept of Use Case to Produce Dynamic Object-Oriented Specifications. In *3rd IEEE Int. Symp. on Requirements Engineering (RE '97)*, pages 54–64. IEEE CS Press, 1997.

[3] G. Denaro and M. Pezzè. Petri Nets and Software Engineering. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 439–66. Springer, 2004.

[4] M. Elkoutbi and R. Keller. Modeling Interactive Systems with Hierarchical Colored Petri Nets. In *Advanced Simulation Technologies Conference 1998*, pages 432–37, 1998.

[5] J. Fernandes, S. Tjell, J. Jørgensen, and O. Ribeiro. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *6th Int. Workshop on Scenarios and State Machines (SCESM 2007), at ICSE 2007*. IEEE CS Press, 2007.

[6] H. Gomaa. A Software Modeling Odyssey: Designing Evolutionary Architecture-Centric Real-Time Systems and Product Lines. In *9th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2006)*, volume 4199 of *LNCS*, pages 1–15. Springer, 2006.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–74, 1987.

[8] M. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[9] K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer, 1992.

[10] K. Jensen, L. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *Software Tools for Technology Transfer*, 2007. In Press. DOI: 10.1007/s10009-007-0038-x.

[11] J. Jørgensen. Addressing Problem Frame Concerns via Coloured Petri Nets and Graphical Animation. In *2nd Int. Workshop on Advances and Applications of Problem Frames (IWAAPF '06), at ICSE 2006*, pages 49–57. ACM Press, 2006.

[12] P. Kruchten. *The Rational Unified Process — An Introduction, Second Edition*. Addison-Wesley, 2000.

[13] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[14] R. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann, 2003.