Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net

João M. Fernandes[1,2] Simon Tjell[2] Jens Bæk Jørgensen[2] Óscar Ribeiro[1]
[1]Dep. Informática / CCTC, Universidade do Minho, Braga, Portugal
[2]Dept. Computer Science, University of Aarhus, Aarhus, Denmark
{jmf,oscar.rafael}@di.uminho.pt, {tjell,jbj}@ daimi.au.dk

Abstract

Using a case study on the specification of an elevator controller, this paper presents an approach that can translate given UML descriptions into a Coloured Petri Net (CPN) model. The UML descriptions must be specified in the form of Use Cases and UML 2.0 Sequence Diagrams. The CPN model constitutes one single, coherent and executable representation of all possible behaviours that are specified by the given UML artefacts. CPNs consitute a formal modelling language that enables construction and analysis of scalable, executable models of behaviour. A combined use of UML and CPN can be useful in several projects. CPN is well supported by CPN Tools and the work we present here is aimed at building a CPN Tools front-end engine that implements the proposed translation.

1 Introduction

The work we present in this paper brings together two modelling languages. The first one is the *UML*, which is the de-facto standard modelling language of the software industry. The second language is *Coloured Petri Nets* (CPNs) [12, 16], which is a well-proven formal modelling language, suitable for describing the behaviour of systems with characteristics like concurrency, resource sharing, and synchronisation. Our objective is to establish an integrated tool environment in which software engineers can take advantage of both common UML tools and *CPN Tools* (www.daimi.au.dk/CPNTools). CPN Tools, which we are developing at University of Aarhus, is a well-proven and well-established tool licensed in more than 4,000 copies, including several hundreds of companies.

Our work is aimed at improving tool support for the main modelling language (CPN) that we use for our research and application on modelling. The results should be seen as work in progress in the sense that we present the design of an extension of CPN Tools; the implementation of the design has not been done yet.

The CPN Tools extension that we are building is, obviously, an instance of the much more general problem of translating scenarios represented in various ways into different kinds of state machines. A number of approaches to this problem have been developed (but none that targets CPN); some of these approaches may well be more mature than ours. Therefore, one of the main reasons why we would like to present our design ideas to the SCESM community is to benefit from feedback from subject matter experts, even-though many of them have preferences for other modelling languages.

As basis for our translation approach, we assume that developers specify the functionality of the system under consideration with use cases (UCs), each of which is described by a set of UML 2.0 sequence diagrams (SDs). For each UC, there should exist at least one SD that represents and describes its main scenario. Other SDs for the same UC are considered to be variations of the main scenario. Our translation approach allows the development team to interactively play or reproduce any possible run of the given scenarios. In particular, the natural characteristics of the CPN language facilitate the representation of the hierarchy and concurrency constructs of SDs.

This paper is structured as follows. Sect. 2 introduces the case study, an elevator controller, and its UCs and SDs. In sect. 3, we explain how to obtain a CPN model from a set of UCs and SDs, including how to translate the operators *opt*, *alt*, *par*, and *loop*, and the *ref* interaction fragment. Sect. 4 discusses related work and the conclusions are drawn in sect. 5, which also points to future work.

This paper assumes that the reader is familiar with the CPN language and SDs.

Sixth International Workshop on Scenarios and State Machines (SCESM'07)
0-7695-2958-5/07 $20.00 © 2007 IEEE

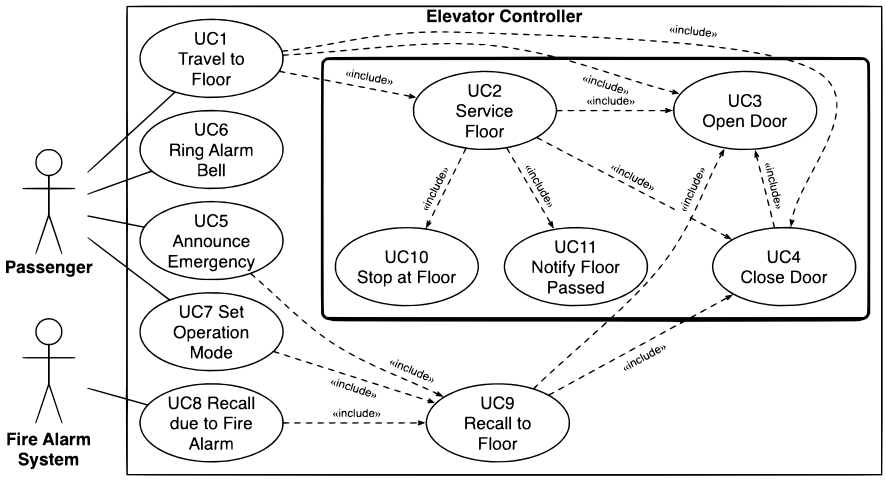

Figure 1. The UC diagram for the Elevator Controller.

2 Case study

This section presents an elevator controller (EC) that is used as a case study in the paper. We use a slight modification of the EC system described in [4]. The elevator system controlled by the EC has two cars, servicing a building with six floors.

The main responsibility of the EC is to control the movements of the cars, which are triggered by passengers pushing buttons. On each floor, there are hall buttons, which can be pushed to call the elevator; a push indicates if the passenger wants to travel up or down. Inside each car, there are floor buttons, which can be pushed to request to be carried to a given floor, and one open door button to force the car door to open. The controller also updates 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 UC diagram for the EC. To focus the discussion, only UC2 and its included UCs (UC3, UC4, UC10, and UC11) are considered. Their (short) descriptions are next given:

UC2 Service Floor moves the elevator car from one origin floor to a destination floor.

UC3 Open Door opens the elevator door.

UC4 Close Door closes the elevator 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.

UC10 Stop Elevator At Floor stops an elevator car at a given floor.

UC11 Notify Floor Passed informs the passengers, inside an elevator car, that a given floor is about to be passed.

Some of the SDs that describe the UCs of the EC are next presented. The actors in the SDs are not those included in the UC diagram, but instead represent the entities (like, sensors and actuators) the EC uses to actually interface with those actors.

Fig. 2 depicts the main scenario of UC2 Service Floor, assuming that the car is at floor F_o and that the next requested floor is F_d.

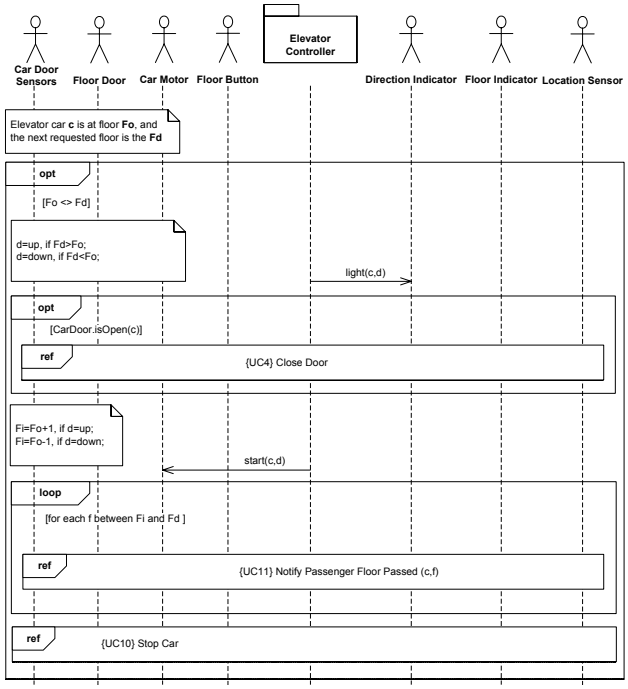

Figure 2. SD for main scenario of UC2 Service Floor.

0-7695-2958-5/07 $20.00

This scenario describes the following behaviour:

1. The passenger in the current floor (F_o) is notified about the direction of the car (specified in the SD by the message "light(c)" from EC to Direction Indicator);
2. The car door is closed, if it is open (specified by the *opt* operator and the *ref* interaction fragment to UC4);
3. The car is moved in direction to floor F_d (specified by the message "start(c,d)" from EC to Car Motor);
4. The passengers inside the car are notified about each floor passed (specified by the *loop* operator and the *ref* interaction fragments to UC11);
5. The car is stopped, when destination floor F_d is reached (specified by the last *ref* interaction fragment to UC10).

The main scenario of UC2 assumes that no requests to stop the car in any intermediate floor exist, while it is moving. Therefore, we need to model this possibility by a new SD (fig. 3). This scenario considers that hall button requests for an elevator must be saved for future processing, in parallel, with the possibility of the car being stopped in an intermediate floor, if a request exists.

This second scenario of UC2 is a variation of the main one. The two SDs for UC2 have a lot in common; they only differ inside the loop, in the parts with a darker background in fig. 3. After the loops, there are apparently some differences, but the existence of a *ref* to UC11 in the SD for the variation is due to the fact that the loop executes one less iteration. This means the behaviour before and after the loop is the same for both scenarios. As we will see, this fact is exploited when obtaining the CPN modules for UCs that are specified by two or more SDs.

3 Translation approach

In this section, we describe the translation approach. We explain the basic idea and describe how to handle the following SD's high-level operators: *opt*, *alt*, *par*, and *loop*. The *ref* interaction fragment is also considered. Some of these ideas were already discussed in the technical report [23], but here we significantly extend the work and introduce some modifications. As examples, the explanation uses the CPN modules that were obtained from the UC diagram (fig. 4) and the two SDs for UC2 (figs. 5 and 6).

3.1 Basic idea

Our 2-step translation generates a CPN model based on a set of UCs ($\{\mathrm{UC}_1, \ldots, \mathrm{UC}_n\}$) and, for each UC_i, a given set of SDs ($\{\mathrm{SD}_{i1}, \ldots, \mathrm{SD}_{im_i}\}$).

The first step in the translation is to construct a CPN module from the n UCs. In fact, one only considers the subset of primary UCs (those connected to actors). For the EC, this CPN module is outlined in fig. 4. This CPN model is a crude, overall description of the possible behaviours specified by the UCs. It includes a so-called substitution transition for each primary UC. A substitution transition is a hierarchical structuring mechanism of CPN that is bound to a separate module of the CPN model. This so-called submodule constitutes a more detailed specification of the behaviour that is represented by the substitution transition on the top-level. The CPN top module permits to choose which primary UC is being executed in a given moment. When there is a token in SP0, one of its output transitions can be fired, which determines the substitution transition that is subsequently entered.

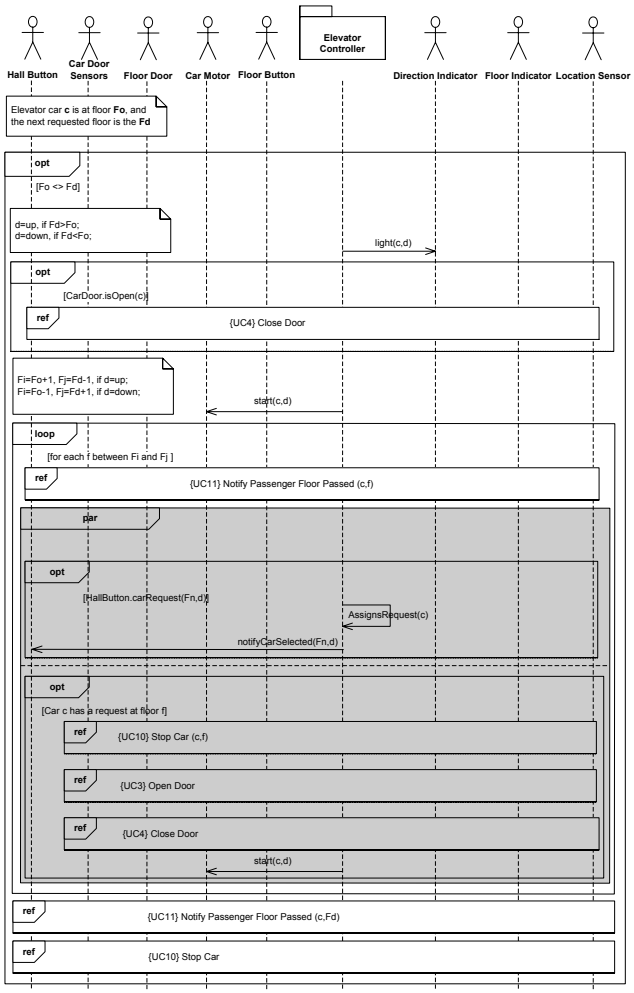

Figure 3. SD for variation of UC2 Service Floor.

0-7695-2958-5/07 $20.00

In the second step, the sub-modules that represent the substitution transitions are constructed, which means that each SD is translated into a CPN module. To explain the main ideas of the SD-to-CPN translation, we will discuss how fig. 5 was obtained from fig. 2.

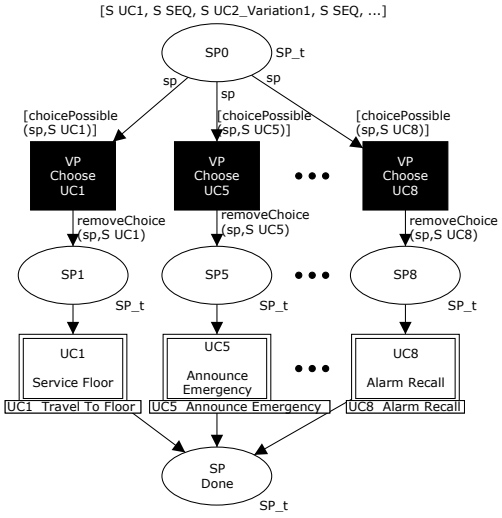

Figure 4. Top module of the CPN model.

The translation of one SD into one CPN module maps each message in the SD into a CPN transition. Thus, a SD with no operators gives origin to a sequence of transitions, which are interleaved with places, to follow the syntax rules of CPNs.

The first message of the SD is "light(c,d)", which translates into the transition "Light Direction Indicator". Next we find an *opt* operator that represents an optional behaviour. Our approach takes into account the alternative execution paths, that are decided at the so-called *variation points* (VPs). They represent execution points, where alternative behaviours can occur. The choice of the path to follow is typically dependent on values from the environment (inputs). Our approach ignores the actual values that are used to accomplish the choice but is able to reflect all alternatives in the CPN model.

The *opt* operator of the example gives origin to a VP, where behaviour can follow one of two alternative paths represented by the two output transitions of SP2. If the car door is assumed to be open, the "VP Car Door Is Open" transition should be fired, which implies that the behaviour represented by UC4 needs to be executed. In the CPN, this maps to the "Close Doors" substitution transition that is described by a different CPN module. When place SP4 is reached, the *opt* operand was either executed or not. The next message is "start(c,d)", which translates to the "Start Motor" transition. This process continues until all messages and operators are considered.

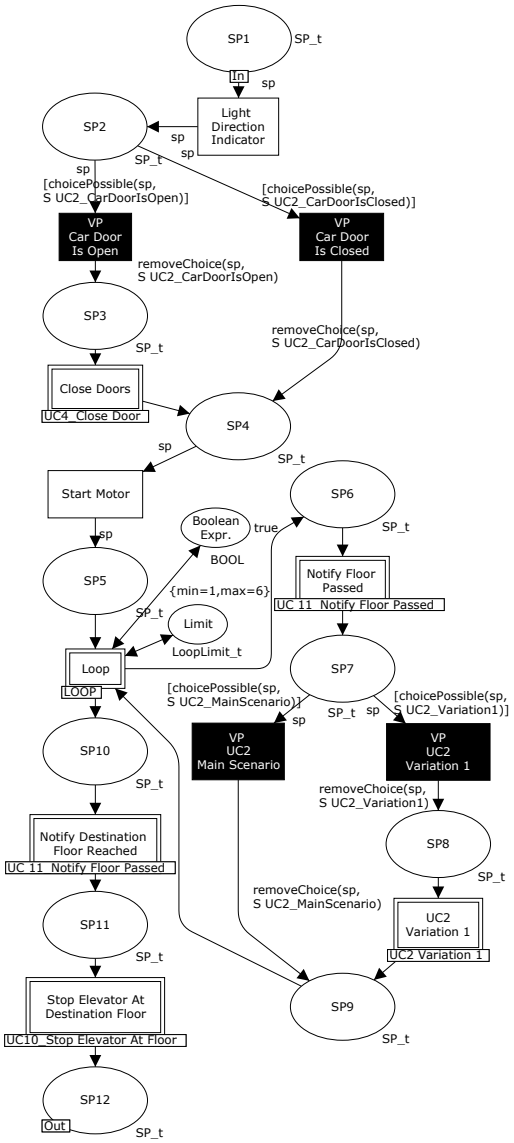

Figure 5. CPN module for UC2's main scenario.

Place SP7 represents a VP where it is possible to execute either UC2's main scenario or its variation, represented by the two SP7's output transitions. In case the variation is selected, by firing the transition on the right, the respective CPN module is entered when the substitution transition "VP UC2 variation 1" is reached.

The main scenario for UC2 also includes a *loop*, represented in the CPN by the "Loop" substitution transition (output transition of place SP5). If a new loop iteration is to be executed, the loop body (CPN parts from SP6 to SP9) is

0-7695-2958-5/07 $20.00

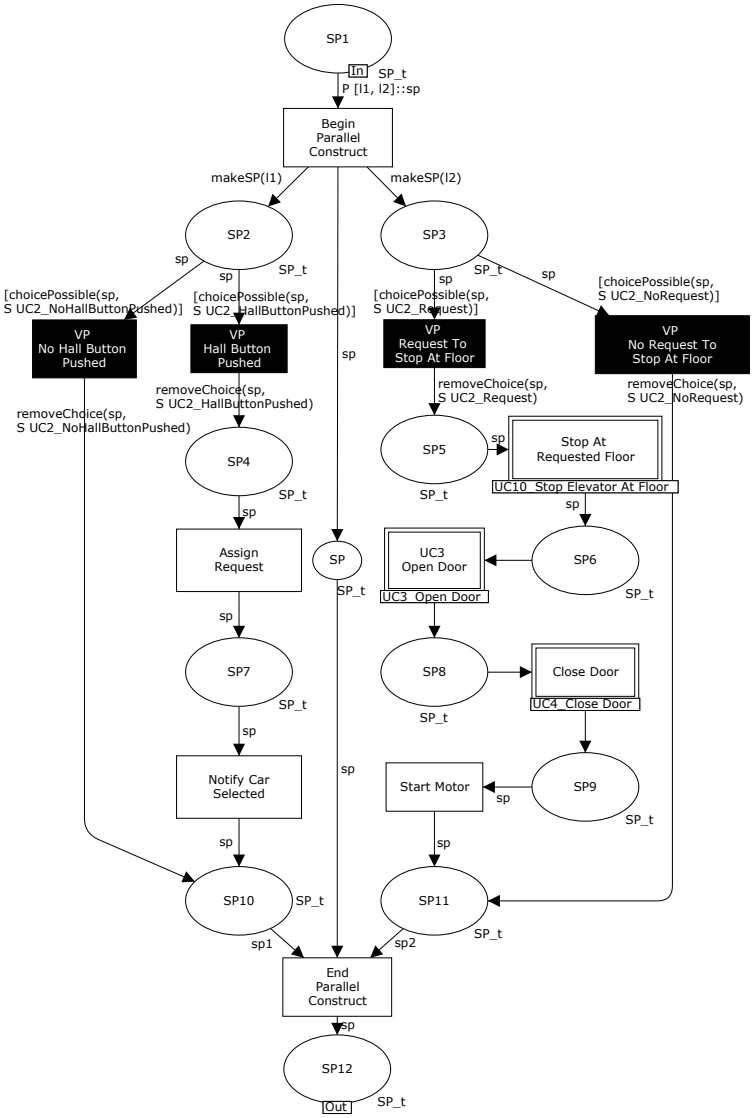

Figure 6. CPN module for UC2's variation scenario.

executed; otherwise the behaviour continues to place SP10. The loop CPN module (sect. 3.2.3) can be used as many times as needed, to translate any SD *loop* operator. This solution promotes reuse and hides complexity.

3.2 Translating the SD operators

This subsection presents how to translate the SD's operators *opt*, *alt*, *par*, and *loop*, and the interaction fragment *ref* into CPN parts.

3.2.1 OPT and ALT

An *opt* operator is like an *if* or *case* statement in programming languages. It represents a behaviour that might or might not be executed, depending on the evaluation of the guard condition. If the guard condition is false, the behaviour is ignored. The translation of an *opt* operand into a CPN part is exemplified in the top part of fig. 5, by the two output transitions of place SP2. The left transition represents the situations where the guard of the operand is evaluated as true and consequently the behaviour specified in the operand in fig. 2 to be executed. The right transition represents the operand having its guard evaluated as false, which implies not executing the operand's behaviour.

An *alt* operator identifies a set of behaviours from which the interaction can choose based on specified criteria. Only one of the alternatives will execute on any one pass through the interaction. The selected operand in the *alt* structure executes only if the guard condition tests true. If there is no guard, the operand always executes when it is selected. The *else* clause of the *alt* combined fragment executes whenever none of the other options is selected. Since an *alt* is a general case of an *opt*, the translation into CPNs basically follows the same rules described for *opt*, but from the same point more alternatives exist to be followed.

When translating an *alt* operator, one must analyse if the guards in all operands cover all possible combinations of the involved inputs. If this is not the case, the CPN model must include an extra branch that represents the situation where none of the guards is evaluated as true. If a guard is always false, the respective operand is never executed and the CPN should not include a branch for it.

3.2.2 PAR

The operator *par* represents a parallel execution of the behaviours of the operands. The occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved. This operator has a natural representation with the CPN language, which supports description of concurrency and parallelism.

The translation of the par operand in fig. 3 (the darker area) into a CPN part is represented in fig. 6 by the transitions 'Begin Parallel Construct' and 'End Parallel Construct'. This fork-join construct allows the behaviour to follow two independent threads, thus modelling true parallelism. It is straightforward to apply this translation to more than two parallel operands by just adding more output places to the fork transition and, similarly, more input places to the join transition.

3.2.3 LOOP

A *loop* operator is like a loop in programming languages and indicates a behaviour that is executed repeatedly. The number of times the contents of the *loop* is executed is given by the *minint* and *maxint* parameters of the operator. The syntax of the *loop* operator is 'loop (minint, maxint)', where maxint can also be infinity (specified by '*'). After the minimum number of iterations has been satisfied, a Boolean expression is tested on each pass. When the Boolean expression tests false, the *loop* ends.

A generic *loop* was modelled as a CPN module (fig. 7). This module can be instantiated as many times as needed (by substitution transitions). The "Initialize Loop" transition starts a counter with value 0, and puts a token in place SP1. This place may enable its output transitions, that represent the choice between executing a new iteration of the *loop* or leaving it. Every time the *loop* is entered the counter is incremented. The counter, the Boolean expression and the *loop*'s limits (minimum and maximum number of iterations) are places that hold the values from the CPN module where the *loop* is instantiated.
The substitution transitions that represent instances of the *loop* module need to identify the *loop*'s limits, a Boolean expression, the two places where the *loop* body starts and ends, and the two places where the *loop* initialises and finishes. Its usage is shown in fig. 5.

3.2.4 REF

SDs can reuse a given SD inside another SD, with the special interaction fragment *ref*. This permits the same advantages as introduced by routines or functions in programming languages, namely abstraction and reuse.

The usage of a *ref* in SDs occurs if a UC includes another one to complete its behaviour. The translation of the *ref* construct to CPN is achieved by including a substitution transition to represent the included UC. For example, the *ref* to UC10 in fig. 2 is modelled by the substitution transition between places SP11 and SP12 in fig. 5.

As already mentioned, the *ref* construct is also used when representing variations of a UC. This is exemplified in fig. 5 by the substitution transition "VP UC2 variation

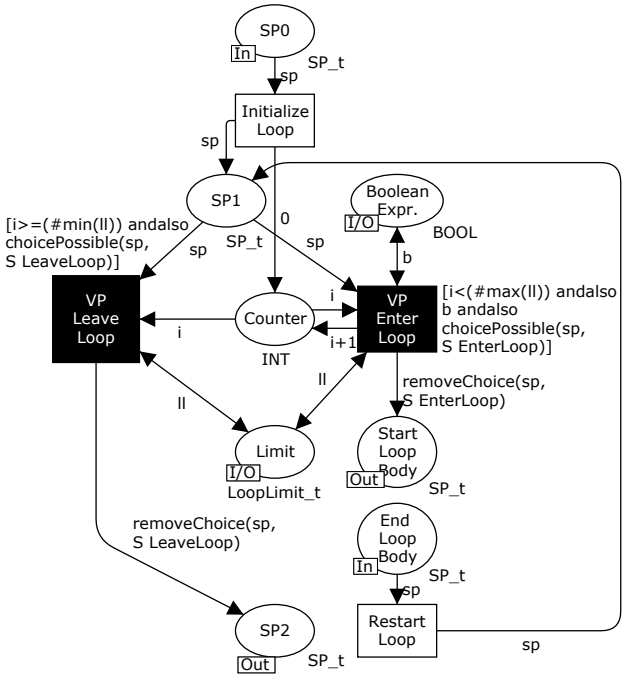

Figure 7. The CPN module for modelling loops.

1" that represents UC2 variation scenario. Again, a substitution transition is included to call the corresponding CPN module. This CPN module should, whenever possible, just model the parts of the variation that differ from the main scenario. This implies that the decision to execute a given variation of a UC is taken inside its execution. Alternatively, we could also consider the choice of the variation at a higher level (namely at the CPN top module). As a consequence, the variations are to be chosen at the top module, at the same level as UCs.

3.3 Execution of the CPN model

The modules of a CPN model constitute one single coherent description of the behaviour specified by the UC diagram. One of the main advantages of a CPN model is that it can be executed. In this subsection, we discuss different ways to execute a CPN model created with our approach.

A CPN model generated by our approach can be executed in three different ways. Two of them are directly supported by CPN Tools and are known as *interactive* and *automatic* simulation. During interactive simulation, the user selects which transitions should be fired, and thereby which execution path to follow. This selection is performed nondeterministically by the tool during automatic simulation. A third possibility is to specify a scenario as a sequence of choices which are determined in advance. This solution allows for some choices to be performed by the user or nondeterministically by the tool, while others are fixed.

3.3.1 Controlled pre-defined execution

In the CPN model, a scenario is specified as a sequence of choices represented by a token containing a list. In the initial marking, this token contains a full list of choices that are considered during the execution of the scenario. While the scenario is being executed in CPN Tools during the simulation of the CPN model, the list is checked to decide the path to take at each VP being encountered. As choices are made, their representations are removed from the list, by the function "removeChoice". This way, the first element of the list is always the next choice to be made (when the relevant VP is reached somewhere during the execution of the scenario).

VPs derive from UCs and SDs and they capture three different types of choices depending on the level of abstraction at which they exist: (1) choice of UC; (2) choice of variations, and (3) choice related to the detailed semantics of the operators in the SDs.

An example of this mechanism can be found in the top module of the CPN model (fig. 4). One transition exists per primary UC at the top level. At this point, UC1 is the choice to be taken at the next VP, since the first element in SP0 initial marking's list is "S UC1". This choice represents the highest level of the specification of a scenario.

After having chosen a UC at the first VP, in some cases it is possible to choose among a collection of variations of this UC. This is the case for UC2, which has a main scenario supplemented by a variation representing that a request button was pushed while the elevator car moves between the origin and the destination floors. This variation is specified by the existence of two SDs for UC2 and is reflected in the CPN model by the VP at place SP7 (fig. 5). The leftmost transition represents the main scenario and the rightmost one represents the variation. If the variation is chosen, the respective module is entered ('UC2 Variation 1'). This case shows how the variation consists of adding more details to the already described main scenario.

Another example is place SP2, which is a VP that can be found in the CPN module for UC2 (top-most part of fig. 5). This example shows how a VP represents a low level choice being made internally in the execution of the UC. In this case, the door can be modelled as being either open or closed, when the VP is reached.

In each VP, the possible choices are modelled by transitions (emphasised in black in the figures). The enabling of these transitions is restricted by guards. In these guards, the list of choices is checked by the function choicePossible to determine which transitions are enabled, i.e., which paths are possible, based on the predetermined choices. The list

of choices is accessible to the transitions in question, because it is being carried around in the model by the tokens that exist in the places forming the main flow of the path.

3.3.2 Use of wildcards

When a scenario is described as a sequence of choices, it is often beneficial to be able to adjust the level of restriction of these choices, i.e., to make some choices free, while fixing others. In the approach described so far, all choices are either free and made by the user during execution of the model, or predetermined by the sequence specified in the list of choices. To cover the gap between these two extrema, wildcards have been introduced to the specification of the scenarios. This is done by including the keywords `ANY` and `SEQ` as accepted elements in the sequences.

These operators have simple semantics as described informally with the next examples. The `ANY` operator replaces a single choice. In the sequence `[a,ANY,b]`, `a` and `b` are fixed choices while the choice in-between these choices is performed freely. After one free choice, the `ANY` operator is consumed from the sequence, leaving only the `b` choice. The `SEQ` operator replaces a sequence of choices until the choice following the `SEQ` operator. In the sequence `[a,SEQ,b,c]`, `a` and `c` are fixed choices while any sequence of choices is allowed to be performed until a `b` choice is performed. At that point, the `SEQ` operator will be consumed from the sequence leaving a sequence containing only the `c` choice.

With the sequence of choices and the wildcard solution, scenarios can be described with a varied level of restriction. Thereby it is possible to describe either specific scenarios or families of scenarios in which some VPs are common and others are different among the members of a given family.

4 Related Work

We discuss in this section approaches that translate scenario-based descriptions into Petri nets (PNs). To our knowledge, this translation has received less attention by researchers than the one into statecharts; cf. that only one approach that uses PNs is considered in [18].

Campos and Merseguer discuss the integration of performance modelling within software development process, based on the translation of almost all UML behavioural models into Generalised Stochastic PNs (GSPNs) [6]. With other colleagues, they explain how: (1) to derive an executable GSPN model from a description of a system, expressed as a set of statecharts [21]; (2) to transform UC diagram to model the usage of the system for each actor [20]; (3) to obtain a performance model representing a concrete execution of the system from SDs and statecharts [3]; and (4) to transform from activity diagrams into GSPNs [19].

Baresi and Pezzè describe how to assign formal semantics to UML by defining translation rules that automatically map UML specifications to high-level PNs [2]. Bokhari and Poehlman propose a technique to translate UML state diagrams to Object Coloured PNs (OCPN), which can be implemented using, for example, CPN Tools. Amorim et al. present an approach to translate LSCs to CPNs for analysing and verifying embedded systems [1].

Shatz and his colleagues propose a translation framework to map UML statecharts and collaboration diagrams into CPNs [24, 11]. Statechart diagrams are first converted to flat state machines. Next, these state machines are translated into Object PNs, which are translatable into behaviourally equivalent CPNs [17]. Collaboration diagrams are used to connect these OPN models and to derive a single CPN for the system under consideration. The obtained CPNs can then be analysed by formal techniques or simulated to infer properties of the system's behaviour.

Pettit and Gomaa describe how to integrate CPNs with object-oriented architecture designs captured by UML communication diagrams. Their method can systematically translate a UML software architecture design into an underlying CPN model, using pre-defined CPN templates based on a set of object behavioural roles [22].

An algorithm to transform Message Sequence Charts (MSCs) into a PN is explained in [15]. The transformation algorithm is exemplified in a railway control system and the obtained PN can then be simulated and analysed using already-known techniques and tools.

Sgroi et al. present how to design communication protocols based on MSCs and PNs in [25]. A protocol is specified as a set of MSCs, each one modeling a scenario, and their relations are captured by a PN. To support analysis and derive an optimized implementation, PNs are used to formally represent the traces of events of the MSCs.

Eichner at al. introduce a formal semantics for the majority of the concepts of UML 2.0 SDs by means of PNs as a formal model [9]. Their approach is focused on capturing behaviour, and simulating and visualising it. An animation environment is being developed, within the P-UMLaut project (www.p-umlaut.de), to relate objects of the simulated world and entities in the UML model. This permits the objects to be animated, using the PN as the main driver. Their work is the one with more similarities with ours (namely on the usage of UML 2.0 SDs), but uses a different PN language (M-nets) and is oriented towards SDs that describe the behaviour of a set of objects. We propose to use the CPN language and the SDs describe parts of the behaviour of a UC.

The play-in/play-out approach [10] aims at specification of reactive systems through an ingenious, intuitive way to automatic generation of executable, formal models from scenarios. It is based on the use of Live Sequence Charts

(LSCs) [7]. Our approach described in this paper is much more mundane, and specifically targeting the CPN modelling language, which is different from LSCs.

5 Conclusions and Future Work

In this paper, we present an approach to translate UML 2.0 SDs into CPN models and illustrate it on an elevator controller case study. According to the criteria proposed in [18], it supports composition by adopting UML 2.0 SDs as the source notation and follows a "GS → GSM" synthesis path, i.e., it transforms global scenarios (GS) into global state machines (GSM), specified in the CPN language.

The work presented in this paper is partly inspired by a joint research project that one of the authors of this paper was involved in together with Nokia Research (Helsinki) in 2003–5. In this project [14], a tool was developed to estimate the worst-case memory usage of interacting software components. The tool applies formal analysis based on CPN. For a given set of interaction scenarios, the tool calculates a state space of a CPN model and finds a path, which corresponds to a worst-case memory usage interleaving of the events in the scenarios. To hide the formal analysis from the users of the tool, IBM Rational Rose is used as front-end to specify scenarios as annotated UML 1.x SDs, and Microsoft Excel is used as back-end to present the analysis results. Nokia sees a possibility in continued use of CPN along the lines set out in the previous project, but now moving from UML 1.x SDs to UML 2.0 SDs. The work presented here contributes to that effort.

The long-term goal we have in focus in continuation of the work described in this paper is to use CPN Tools as a vehicle for requirements engineering. Our main goal is to build graphical animations on top of CPN models. This has already been done in a number of projects (e.g., [13]), but it has turned out that for a broader applicability, it is a problem that the approach has not been properly integrated with UML. With the work we present in this paper, we take a step towards making a tighter connection between UML and CPN models, aimed at the use in requirements engineering.

In general, capturing the right requirements from the end users and clients of the system is one of the main problems encountered in software development projects. This demands ways of effective communication among developers and users. Creating models, like UML models or CPN models, can be a help, but models of these kinds are often uninviting for many stakeholders [8, 26]. In particular, UML 2.0 SDs are much more complex than UML 1.4 SDs, due to the usage of high-level operators, which amplifies the comprehension problem by non-technical users.

One of the solutions that can improve the comprehension of the system's dynamic behaviour is, indeed, to use animation techniques, thereby enabling involvement of non technically-minded stakeholders. This can, of course, give important, early feedback about the intended system. Some authors report that the use of animation greatly boosts the understanding of the system behaviour when compared with static models (such as SDs) [5].

Among our next steps is the automation of the translation approach. We expect our tool to be useful in projects that we are currently working on establishing. An example is a project with Torrestir, a Portuguese truck company, where UC diagrams and SDs will be developed by project participants and our aim is to use these SDs as basis for further requirements engineering using our CPN-based approach.

The basis for our approach is a set of UCs and for each UC, a set of SDs, which constitute a description of a set of behaviours; for the EC case study, possible ways in which the elevator system can behave. Thus, in the near future we plan to study how to handle the simultaneous execution of more than one use case instance (for example, two instances of UC3 and one instance of UC7 of the EC system). Although we do not have currently clear answers for this issue, the CPN language seems to be well-suited for that purpose, since it supports naturally true parallelism. This probably requires some mechanisms to be added to the CPN, namely for synchronising the execution of the behaviours and to control the usage of the shared resources.

We do not plan to use CPN models directly to implement software systems. Our focus is more narrowly on requirements elicitation, validation, and specification; not on turning requirements into implementations. One reason for this is that we know only a few projects in which PN models have been used as basis for a running implementation of a system. There seems to be a larger gap between PN models, including CPN models, and implementations than there is between, e.g., statecharts or UML state machines and implementations. This is probably one of the main reasons that proposals to use PNs is often met with some reluctance in the software engineering community. PNs, which have their origins as a theoretical model for concurrency, are suffering from image problems and from lack of demonstration of usefulness in actual software engineering.

With our narrow focus on early requirements, we hope to contribute to improve the situation, because this is an area where we believe in the usefulness of CPN in relation to software engineering. The rich and elaborated state concept of CPN models makes it easier to represent requirements and relevant environment properties.

6 Acknowledgements

This work was done while João M. Fernandes was on a sabbatical leave at University of Aarhus, partially supported by Fundação para a Ciência e a Tecnologia (FCT), under grant SFRH/BSAB/607/2006. The authors acknowl-

edge João P. Barros and Peter G. Larsen, who have provided very important and useful feedback on an earlier version of this paper.

References

[1] L. Amorim, P. Maciel, M. Nogueira, R. Barreto, and E. Tavares. Mapping Live Sequence Chart to Coloured Petri Nets for Analysis and Verification of Embedded Systems. *SIGSOFT Software Engineerign Notes*, 31(3):1–25, 2006.

[2] L. Baresi and M. Pezzè. On Formalizing UML with High-Level Petri Nets. In *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 276–304. Springer, 2001.

[3] S. Bernardi, S. Donatelli, and J. Merseguer. From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. In *3rd Int. Workshop on Software and Performance (WOSP 2002)*, pages 35–45. ACM Press, 2002.

[4] R. M. Blanco. Requirements Specification for an Elevator Controller. Technical report, School of Computer Science, University of Waterloo, Canada, 2005.

[5] E. Burd, D. Overy, and A. Wheetman. Evaluating Using Animation to Improve Understanding of Sequence Diagrams. In *10th Int. Workshop on Program Comprehension (IWPC 2002)*, pages 107–13. IEEE CS Press, 2002.

[6] J. Campos and J. Merseguer. On the Integration of UML and Petri Nets in Software Development. In *27th Int. Conf. on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2006)*, volume 4024 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2006.

[7] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001.

[8] N. Dulac, T. Viguier, N. Leveson, and M.-A. Storey. On the Use of Visualization in Formal Requirements Specification. In *IEEE Joint Int. Conf. on Requirements Engineering (RE 2002)*, pages 71–80. IEEE CS Press, 2002.

[9] C. Eichner, H. Fleischhack, R. Meyer, U. Schrimpf, and C. Stehno. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In *SDL 2005: Model Driven Systems Design*, volume 3530 of *Lecture Notes in Computer Science*, pages 133–48. Springer, 2005.

[10] D. Harel and R. Marelly. *Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[11] Z. Hu and S. M. Shatz. Mapping UML Diagrams to a Petri Net Notation for System Simulation. In *16th Int. Conf. on Software Engineering & Knowledge Engineering (SEKE 2004)*, pages 213–9, 2004.

[12] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1-3*. Monographs in Theoretical Computer Science. EATCS Series. Springer, 1992-97.

[13] J. B. Jørgensen and C. Bossen. Executable Use Cases as Links Between Application Domain Requirements and Machine Specifications. In *3rd Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2004)*, pages 8–13. IEE, 2004.

[14] J. B. Jørgensen, S. Christensen, A.-P. Tuovinen, and J. Xu. Tool Support for Estimating the Memory Usage of Mobile Phone Software. *Software Tools for Technology Transfer*, 8(6):531–45, 2006.

[15] O. Kluge, J. Padberg, and H. Ehrig. Modeling Train Control systems: From Message Sequence Charts to Petri Nets. In *Formale Techniken für die Eisenbahnsicherung (FORMS 2000)*, pages 25–42. Fortschritt-Berichte VDI, 2000.

[16] L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *Software Tools for Technology Transfer*, 2(2):98–132, 1998.

[17] C. A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *16th Int. Conference on the Application and Theory of Petri Nets (ICATPN 1995)*, volume 935 of *Lecture Notes in Computer Science*, pages 278–97. Springer, 1995.

[18] H. Liang, J. Dingel, and Z. Diskin. A Comparative Survey of Scenario-based to State-based Model Synthesis Approaches. In *Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2006)*, pages 5–12. ACM Press, 2006.

[19] J. P. López-Grao, J. Merseguer, and J. Campos. From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering. In *4th Int. Workshop on Software and Performance (WOSP 2004)*, pages 25–36. ACM Press, 2004.

[20] J. Merseguer and J. Campos. Exploring Roles for the UML Diagrams in Software Performance Engineering. In *3rd Int. Conf. on Software Engineering Research and Practice (SERP 2003)*, pages 43–7. CSREA Press, 2003.

[21] J. Merseguer, J. Campos, S. Bernardi, and S. Donatelli. A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In *6th Int. Workshop on Discrete Event Systems (WODES 2002)*, pages 295–302. IEEE CS Press, 2002.

[22] R. G. Pettit and H. Gomaa. Modeling Behavioral Design Patterns of Concurrent Objects. In *28th Int. Conf. on Software Engineering (ICSE 2006)*, pages 202–11. ACM Press, 2006.

[23] O. R. Ribeiro and J. M. Fernandes. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In *7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006)*, pages 237–56, 2006.

[24] J. Saldhana and S. M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. In *Int. Conf. on Software Engineering & Knowledge Engineering (SEKE 2000)*, pages 103–10, 2000.

[25] M. Sgroi, A. Kondratyev, Y. Watanabe, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of Petri Nets from Message Sequence Charts Specifications for Protocol Design. In *Design, Analysis, and Simulation of Distributed Systems (DASD 2004)*, pages 193–9, 2004.

[26] M. K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the Readability of State-Based Formal Requirements Specification Languages. In *24th Int. Conf. on Software Engineering (ICSE 2004)*, pages 33–43. ACM Press, 2002.