Some Rules to Transform Sequence Diagrams into Coloured Petri Nets [⋆]

Óscar R. Ribeiro and João M. Fernandes

{oscar.rafael, jmf}@di.uminho.pt
Dept. of Informatics, University of Minho, Portugal

Abstract. This paper presents a set of rules that allows software engineers to transform the behavior described by a UML 2.0 Sequence Diagram (SD) into a Colored Petri Net (CPN). SDs in UML 2.0 are much richer than in UML 1.x, namely by allowing several traces to be combined in a unique diagram, using high-level operators over interactions. The main purpose of the transformation is to allow the development team to construct animations based on the CPN that can be shown to the users or the clients in order to reproduce the expected scenarios and thus validate them. Thus, non-technical stakeholders are able to discuss and validate the captured requirements. The usage of animation is an important topic in this context, since it permits the user to discuss the system behavior using the problem domain language. A small control application from industry is used to show the applicability of the suggested rules.

1 Introduction

Although complex systems are, by their nature, hard to build, the problem can be ameliorated if the user requirements are rigorously and completely captured. This task is usually very difficult to complete, since clients and developers do not use the same vocabulary to discuss. For behavior-intensive applications, this implies that the dynamic behavior is the most critical aspect to take into account. This contrasts with database systems, for example, where the relation among data types is the most important concern to consider. A scenario is a specific sequence of actions that illustrates behaviors, starting from a well defined system configuration and in response to external stimulus. Petri nets are used to formalize the behavior of some component, system or application, namely those that have a complex behavior. Since Petri nets are a formal model, they do not carry any ambiguity and are thus able to be validated.

This paper proposes a set of rules that allow software engineers to transform requirements, expressed as a UML 2.0 SD into an behaviorally equivalent CPN. The main purpose of the transformation is to generate a CPN [1] that is akin to be animated (with the mechanism available in CPN-Tools) and thus understood by the users. The synthesized CPN that can be shown (i.e. animated) to

[⋆] This work has been supported by the grant with reference SFRH/BD/19718/2004 from *"Fundação para a Ciência e Tecnologia"*

the users or the clients in order to reproduce the expected scenarios and thus validate them. Thus, non-technical stakeholders are able to discuss and validate the captured requirements. The usage of animation is an important topic in this context, since it permits the user to discuss the system behavior using the problem domain language, which they are supposedly familiar with.

The paper is organized as follows. In section 2, the SDs of UML 2.0 are introduced. Section 3 presents some rules of how to translate a SD into a behaviorally equivalent CPN. In section 4, the result of applying the rules presented in previous section to the case study of an industrial reactor system. In Section 5 is presented a discussion of the related work. Section 6 presents the conclusions of this work and some possible directions for the future work.

2 UML Diagrams for interaction

The introduction of UML 2.0 standard changed almost every sort of things in previous versions designated by UML 1.0.

The dynamic part of the system can be specified in UML 2.0 through various behavioral diagrams, such as: activity diagrams, sequence diagrams and state machines diagrams. These diagrams use behavioral constructs, namely activities, interactions, and state machines.

Interactions are a mechanism for describing systems, which can be understood and produced, at varying level of detail. Usually, interactions do not tell the complete story, when they are produced by designers or by computer systems, because normally some other legal and possible traces are not contained within the described interactions. There are some exceptions where the project request that all possible traces of a system shall be documented through interactions.

An interaction is formed by lifelines and messages between them, that sequence is important to understand the situation. Although data may be also important, its manipulation is not the focus of interactions. Data is carried by the messages, and stored in the lifelines, and can be used to decorate the diagrams.

SDs are the most common interaction diagram defined by the UML [2], that focus on the message interchange between a number of lifelines. Communication diagrams show interactions through an architectural view where the arcs between the communicating lifelines are decorated with description of the passed messages and their sequencing. Interaction overview diagrams are a variant of activity diagrams that define interactions in a way that promotes overview of the control flow, these diagrams can be seen as a high-level structuring mechanism that is used to compose scenarios through sequence, iteration, concurrency or choice. There are also optional diagram notations such as timing diagrams and interaction tables. In this work, we concentrate on SDs.

In the UML 2.0 new notions for SDs are introduced to treat iterative, conditional and various other control of behavior. The old iteration markers and guards on messages have been dropped from SDs.

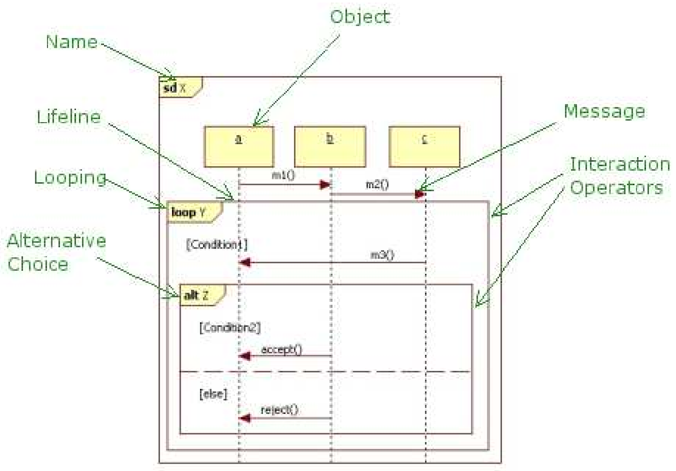


Fig. 1: An example of a UML 2.0 SD

Fig. 1 shows an example of a SD. A SD is enclosed in a frame and includes a pentagon in the upper left handed corner with the keyword `sd` followed by a label identifying the SD.

There are several possible operators, whose meaning is described informally in the UML 2.0 Superstructure specifications [2]:

– **sd:** Indicates the principal frame of the sequence diagram;
– **ref:** references another fragment of interaction;
– **seq:** indicates the weak sequencing of the operands in the fragment, which is select by default. The weak sequencing maintain the order inside each operand, and the events on different operands and different lifelines may occur in any order.
– **strict:** specifies that messages in the fragment are fully ordered;
– **alt:** specifies that the fragment represents a choice between two possible behaviors. There is a guard associated with the fragment, its evaluation define which of choices is executed;
– **par:** indicates that the fragment represents a parallel merge between the behaviors of the operands;
– **loop:** indicates an interaction fragment that shall be repeated some number of times. This may be indicated using a guard condition, and it is executed until the guard evaluates to false.

UML 2.0 provides two kinds of conditions in SDs, namely interaction *constraints* and *state invariants*. An interaction *constraint* is a boolean expression shown in square brackets covering the lifeline where the first event will occur, positioned above that event inside an interaction operand. A *state invariant* is a constraint on the state of an instance, and is assumed to be evaluated during run time immediately prior to the execution of the next event occurrence. Notationally, state invariants are shown as a constraint inside a state symbol or in curly brackets, and are placed on a lifeline.

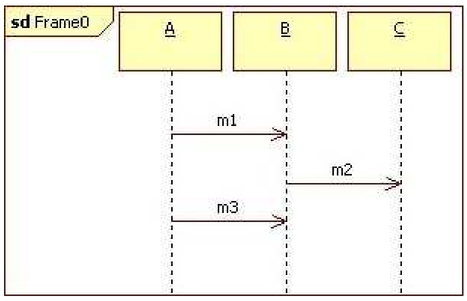
In the previous versions of UML it was not possible to express that, at any time, a specific scenario should not occur. In the UML 2.0 negative behavior (i. e. invalid traces) can be specified using the new operator **neg**. Currently, this operator is not considered in this work.

3 Transforming SD Operators into CPNs

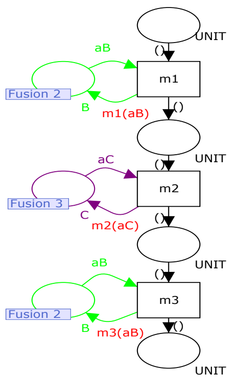
In this section we show how to translate some of the high-level operators available in the UML 2.0 SDs, into a behaviorally equivalent CPN. To accomplish this, we explain the semantics of the operator, we describe in an informal way how the transformation is achieved, and additionally we show the result of applying these ideas to some illustrative examples. We restrict our study to the following high-level operators: `strict`, `seq`, `par`, `loop` and `alt`. Operators like `neg`, `assert`, `critical` are not considered by now.

First of all we look to *InteractionFragments* without any of the high-level operators. An *InteractionFragment* is a set of *Lifelines*, each of which has a sequence of *EventOccurences* associated with it.

We consider a semantic for SD with a order relation between messages such that the emission requires the reception of the preceding message.



(a) A UML 2.0 SD without high-level operators



(b) The obtained CPN

Fig. 2: Example of transform a SD without high-level operators

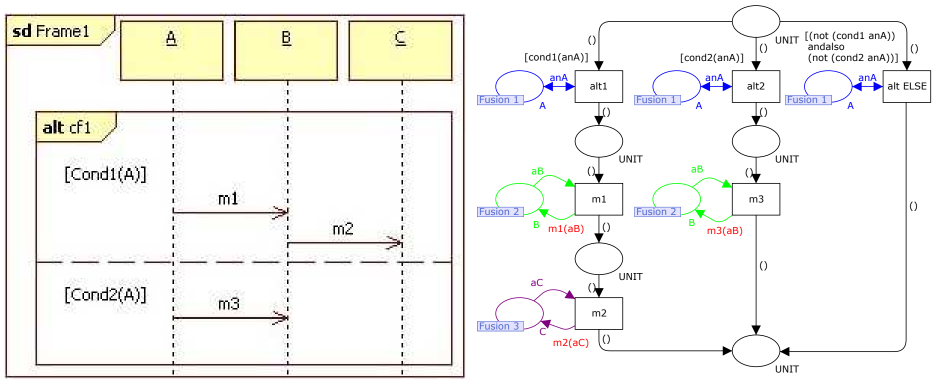
The SD presented in Fig. 2a represents an interaction without high-level operators. There are three *Lifelines* and three messages between them. The obtained CPN (see Fig. 2b) associates a transition for each message in the SD. In this way an execution of a message is represented by the firing of its corresponding transition in the CPN. There are places to guarantee the order between the firring of transitions, and other places to represent the object which the message changes when executing. When firing a transition, a function is applied to the object in the place. This function is a representation of the changes made in the object of the lifeline in the message's destination. To represent the guards in the SD we use a transition guarded by a conditions over the object representation. When there are more than one message in the same point of a lifeline we consider a unique transition which includes all the messages.

Let us consider *CombinedFragments* with high-level operators.

3.1 Alternative Choice

The choice of behavior is represented by a *CombinedFragment* with the *interactionOperator* `alt`. Each operand of `alt` has an associated guard, which is evaluated when choosing the operand to be executed. No more than one operand will be chosen and in this work we assume that the guards must be disjoint. When one of the operands has its guard evaluated to true, the interaction associated with this operand is considered. The empty guard is by default evaluated to true. The operand guarded by `else` means that the guard is evaluated to true when none of the guards in the other operands is evaluated to true. In the case that none of the operands' guards are evaluated to true (this means that there are no `else` and empty guards) none of the operands are executed.

The SD in Fig. 3a is transformed into the CPN in Fig. 3b. Each operand in SD is transformed in a sequential branch. All sequential branchs begin in a common input place and end a common output place.



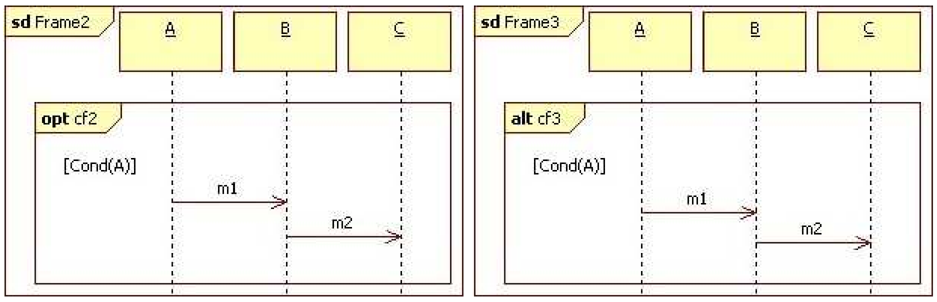
(a) A UML 2.0 SD (b) The obtained CPN

Fig. 3: Example with the alternative choice operator (`alt`)

Please notice that in this case there is no `else` guard, and thus when none of the guards is evaluated as true, no operand is executed. In terms of CPNs this is represented by the rightmost part of the CPN where the "`alt ELSE`" transition condition is the negated disjunction of all other guards. The other branches are guarded by the same condition as in SD and describe the same sequence.

3.2 Optional

The optional operator, represented by *InteractionOperator* `opt`, can be seen as an alternative choice with only one *Operand*, whose guard is not the `else` (see Fig. 4). With this similarity, we can apply to the optional operator the same general translation scheme used for alternative choice.



(a) A UML 2.0 SD (b) The corresponding UML 2.0 SD with `alt`

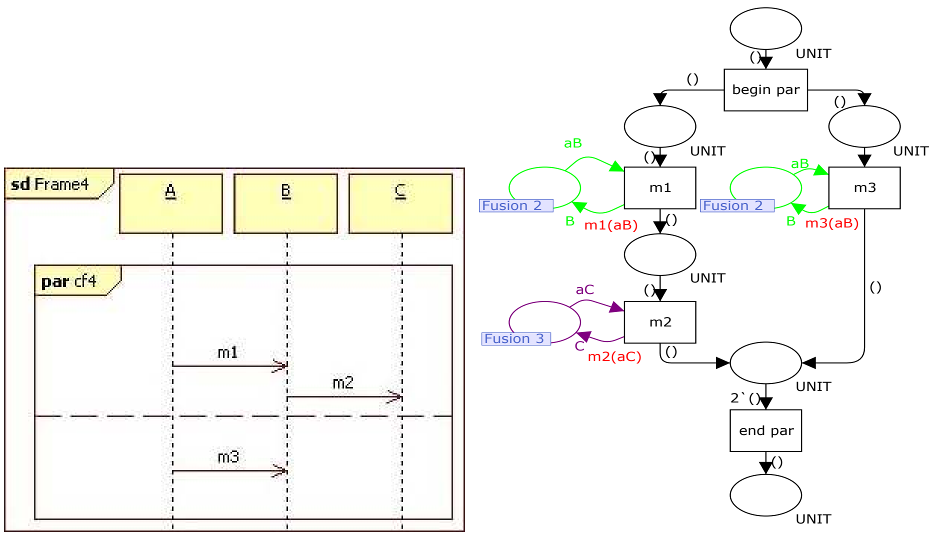
Fig. 4: The option operator (`opt`) expressed by an alternative choice

3.3 Parallel Composition

The parallel merge between two or more behaviours is represented by a *CombinedFragment* with the *interactionOperator* `par`. Keeping the order imposed within operands, *EventOccurrences* from different operands can be interleaved in any way. The SD in Fig. 5a is transformed into the CPN in Fig. 5b. The obtained CPN has two additional transitions to control the interleaving of behaviors. The transition "`begin par`" creates two branches (one for each operand) introducing a token into the two output places, in this way we obtain the interleaving between the transitions of each branch. The transition "`end par`" wait for the execution of all created branches, because it is enabled only when its input place has a number of token equal to the number of created branches.

3.4 Weak Sequencing

When using the *InteractionOperator* `seq` the corresponding *CombinedFragment* represents a weak sequencing between the behaviors of the operands. The or-



(a) A UML 2.0 SD (b) The obtained CPN

Fig. 5: Example with the parallel composition operator (*par*)

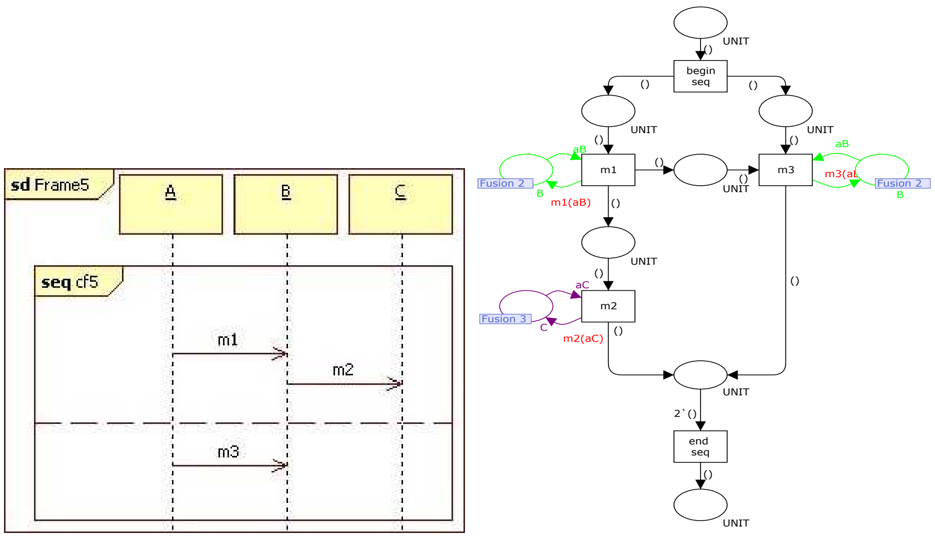
dering of *EventOccurrences* within each of the operands are maintained in the result. *OccurrenceSpecifications* on different lifelines from different operands may come in any order. *OccurrenceSpecifications* on the same lifeline from different operands are ordered such that an *EventOccurrence* of the first operand comes before that in the second operand.

In Fig. 6a we have an example of a SD with `seq` operator. The messages `m1` and `m3` have the *EventOccurence* in the same *Lifeline*, and in the first operand, after the message `m1` we have the message `m2`. Thus, message `m1` must occur before messages `m3` and `m2`.

To construct a corresponding CPN to a SD with the `seq` operator, we first consider the CPN for the parallel composition between the operands, and after that we impose some more order between transitions in different branches. The CPN in Fig. 6b is obtained from the CPN in Fig 5b changing the name of transitions "`begin par`" and "`end par`") to "begin seq" and "end seq", adding the place between transitions "`m1`" and "`m3`" and the corresponding arcs to complete the connection.

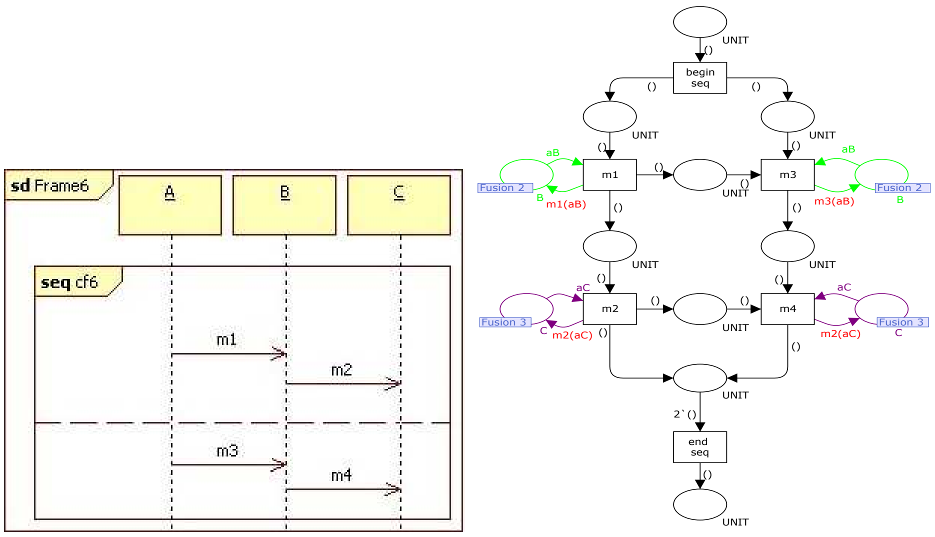
The SD in Fig. 7a is another example using the operator `seq`. The corresponding CPN is presented in Fig. 7b.

There are some particular cases using this operator. If the *EventOccurrence* of the last message from the first operand is in the same *Lifeline* as the first message of the second operand, we have a sequential order between all the messages in the operands. If none *EventOccurrence* of messages is in the same lifeline we have a parallel composition between the operands.



(a) A UML 2.0 SD (b) The obtained CPN

Fig. 6: Example with the weak sequencing operator (*seq*)



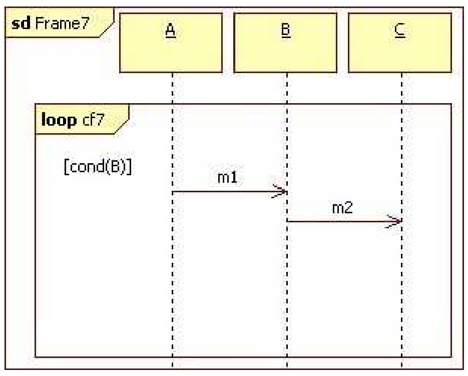
(a) A UML 2.0 SD (b) The obtained CPN

Fig. 7: Another example with the weak sequencing operator (*seq*)

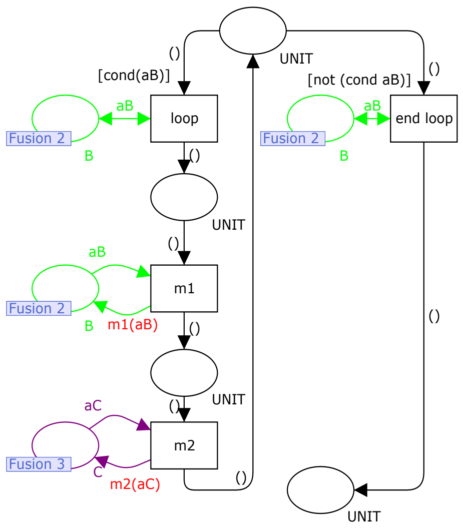
3.5 Looping

The `loop` *InteractionOperator* represents the iterative application of the operand in the *CombinedFragment*. This iterative application can be controlled by a guard or by a minimum and maximum number of iterations.

Given the CPN for the operand inside the `loop`, we add two transitions: "`loop`" and "`end loop`". These two transitions have the same input place. Transition "`loop`" is enabled if the condition (guard for `loop` operator) evaluates to true, and its output place is the input place for the operand's CPN. The transition "`end loop`" is enabled when the condition evaluates to false and its output place is used as connection to the end of the `loop` operator. In Fig. 8 we have an example with the `loop` operator.



(a) A UML 2.0 SD



(b) The obtained CPN

Fig. 8: Example with the looping operator (*loop*)

4 Validation of the Rules

To validate the proposed transformation rules we plan to apply them to several case studies, so that we can also evaluate their practical usefulness. Currently we are using an industrial reactor as a case study. In this chapter we show two CPNs for the reactor: one obtained directly from the requirements (or more precisely adapted from a PN-based specification) and another one obtained from a SD using the proposed translation rules.

4.1 A Case Study: Industrial Reactor

The industrial reactor system consists in a reactor that controls the filling of a tank. It was used in previous works [3–5].

A plant of the reactor system is presented in Fig. 9. The system has two storage vessels, called `SV1` and `SV2`, each of them has a valve (`openSV1` and `openSV2`) to control the exit of liquid. Downside of each storage vessel there is a measuring vessel (`MV1` and `MV2`). A measuring vessel has the same structure as the storage vessel plus two sensors, one indicating when it is full and another when it is empty.

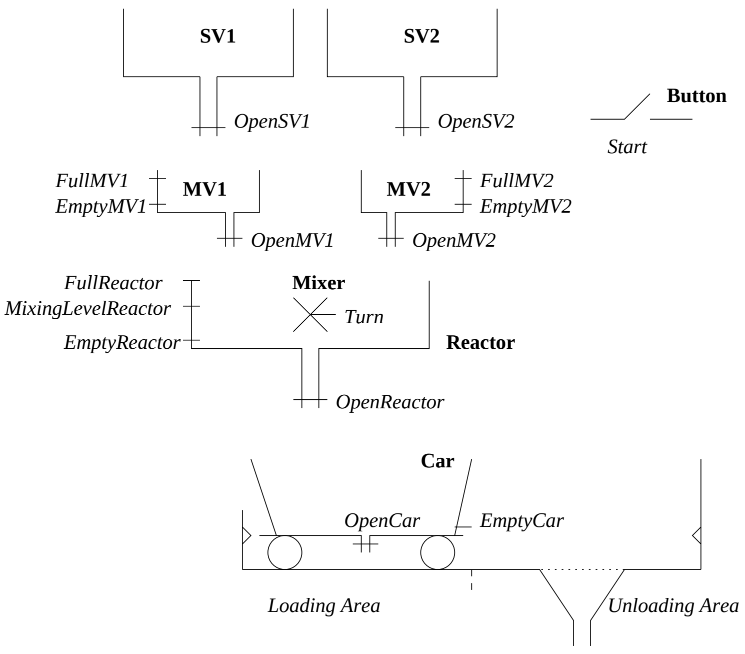


Fig. 9: The environment of industrial reactor system

The `Reactor` is fed with two kinds of liquids from measuring vessels `MV1` and `MV2` which draw from storage vessels `SV1` and `SV2`. After the reaction between the liquids is complete, the reactor is discharged into catch vessel named `Car`. When the `Reactor` is empty the process product is transported using carriage `Car`. To ensure complete reaction the process liquid in the reactor is agitated by stirrer `Mixer`.

When the push button `Start` is pressed the valves `OpenSV1` and `OpenSV2` are opened and measuring vessels `MV1` and `MV2` are refilled until a high-level condition `FullMV1` (`FullMV2`) is sensed. After that, `OpenSV1` (`OpenSV2`) is closed.

The reactor is filled with a liquid input control valves `OpenMV1`, `OpenMV2` and a product discharge valve `OpenReactor`. At the start of a reaction cycle charges of process, liquids are delivered into the reactor from the measuring vessels `MV1` and `MV2`. The valves `OpenMV1`, `OpenMV2` are opened while this proceeding the reactor stirrer may start (`Turn`), when the level in the reactor is higher than `MixingLevelReactor`. When a low level (`EmptyMV1` in `MV1`, `EmptyMV2` in `MV2`) is sensed the valves `OpenMV1` an `OpenMV2` must be closed and the reactor is emptied (`OpenReactor`). After discharging the reactor (`EmptyReactor`) product is transported by using carriage which may move right (`GoUnloadingArea`) or left (`GoLoadingArea`).

4.2 A Manual CPN model

A model of the industrial reactor using High-Level Petri Nets was presented in [4], where a shobi-PN (Synchronous, Hierarchical, Object-Oriented and Interpreted Petri Net) model of an industrial reactor control system is considered as a case study to illustrate the model's applicability and capabilities. The shobi-PN model is an extension to SIPNs (Synchronous and Interpreted Petri Nets) [5]. The model of shobi-PN includes the same characteristics as the SIPN model, in what concerns to synchronism and interpretation, and adds to functionalities by supporting object-oriented modeling approaches and new hierarchical mechanism, in both the control unit and the plant. We have done a translation of SIPN models into PROMELA code to improve the analysis methods for SIPNs [6]. We used the SIPN model of industrial reactor as a case study to validate our approach.

Based on the shobi-PN model of reactor system presented in [4] we created the CPN model in Fig. 10.

The objects are represented by record colors, and the methods are represented by functions on the the object's color, e.g. the storage vessel object and the method to open a storage vessel is defined by the following CPN-ml code:

```
colset StorageVessel =
         record  id       : INT *
                 isOpen   : BOOL*
                 capacity : INT ;

fun openStorageVessel (sv:StorageVessel)
               = StorageVessel.set_isOpen sv true;
```

In this model we have some more pipelining between tasks of the system, than in the shobi-PN model. For example, it is possible to be emptying the storage vessels while the the car is going to the unloading area.

Notice that the tokens associated to each instance of an object is used to control the behavior of the CPN model. The place `anti-place` is only used to restrict the firing of transition `t1`. To simulate the system behavior we create a CPN to represent the environment of reactor.

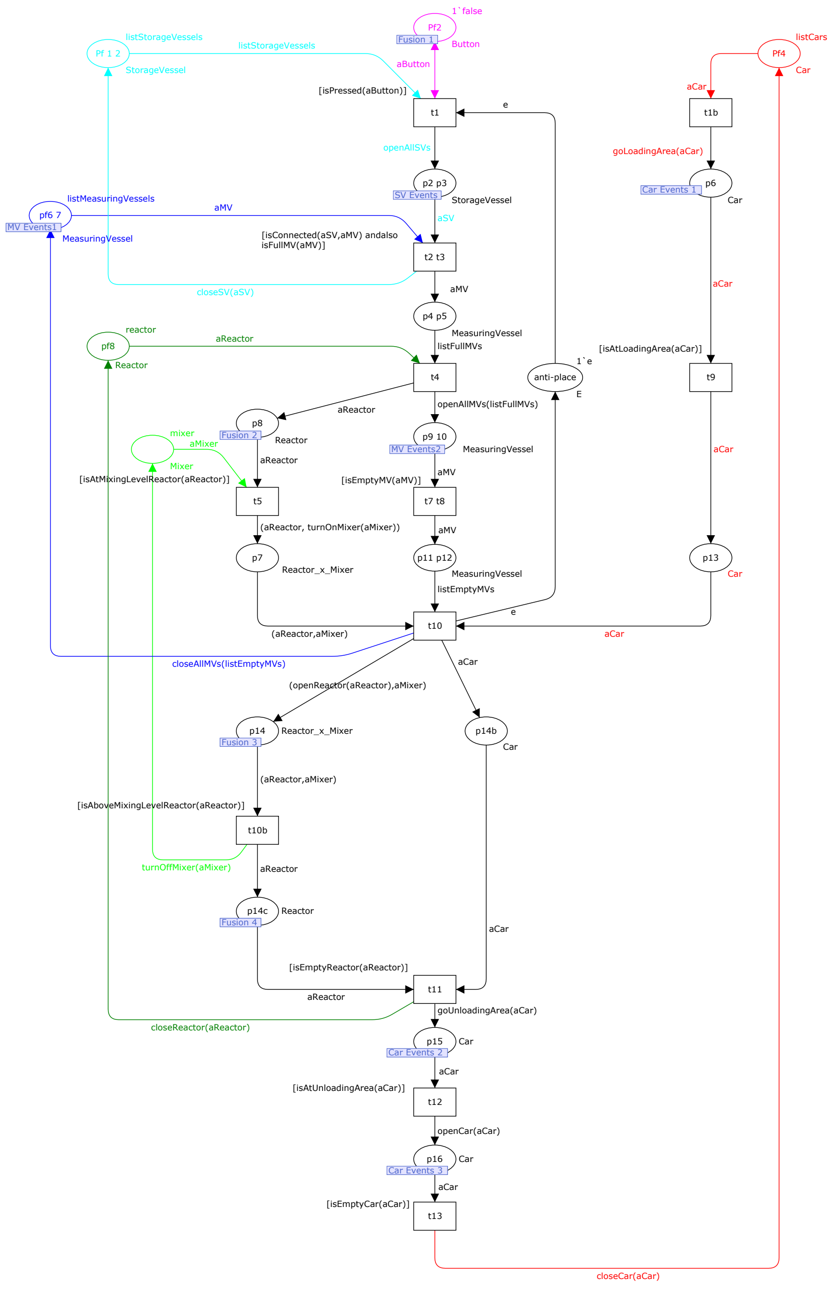


Fig. 10: A CPN model for reactor system

4.3 A CPN model from a SD

This subsection presents a SD for some scenarios of reactor system's usage, and a CPN model obtained from the SD through the rules presented in section 3.

The reactor system is textually described in subsection 4.1, and in subsection 4.2 there is a CPN model which defines its behavior. Taking into account these two exercises of analyzing the reactor system we construct the SD in Fig. 11 to represent the handled scenarios. This SD uses high-level operators, namely the `ref` to point to another two SDs: "Preparing Car" (see Fig. 12) and "Vessels Behavior" (see Fig. 13).

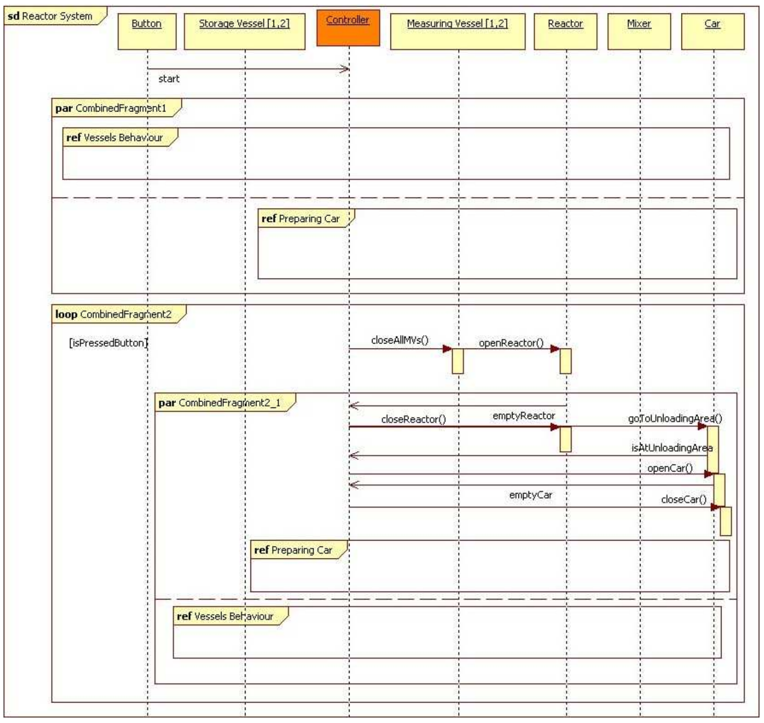


Fig. 11: A SD describing some scenarios of using the reactor system

To transform this SD we firstly apply the rules to the fragments with one high-level operator. After that we compose the obtained CPNs into a hierarchical CPN. We put each SD pointed by `ref` into a subpage. Fig. 14 shows the CPN obtained from the SD in Fig. 12, where we can find transitions which are links to a CPN in a subpage. The subpage `Vessels Behavior` is presented in Fig. 14, which corresponds to the SD presented in Fig. 12.

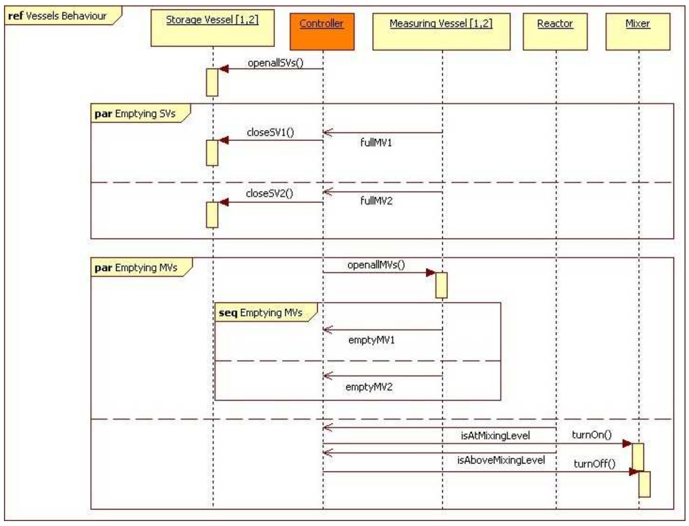


Fig. 12: A SD describing the behavior of vessels

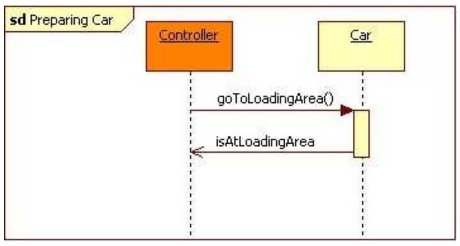


Fig. 13: A SD describing the preparation of car

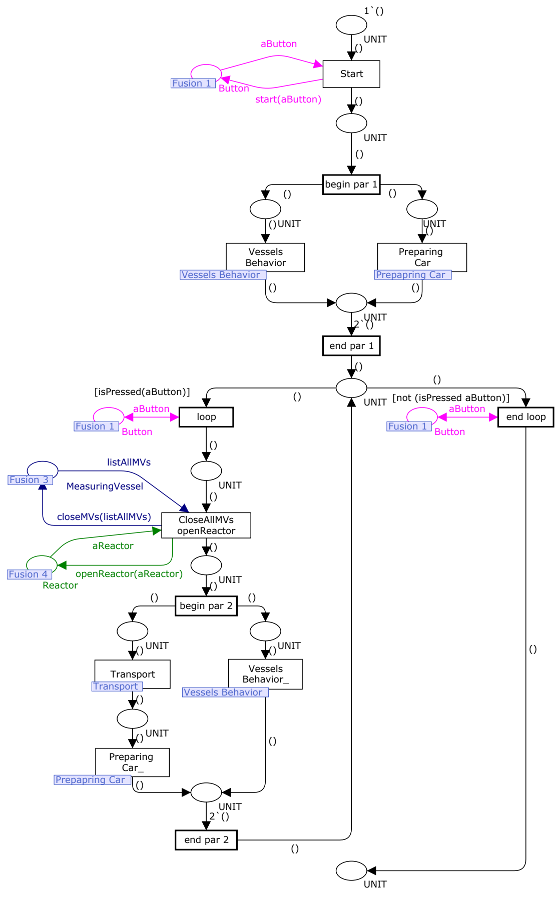


Fig. 14: CPN from the SD presented in Fig. 11

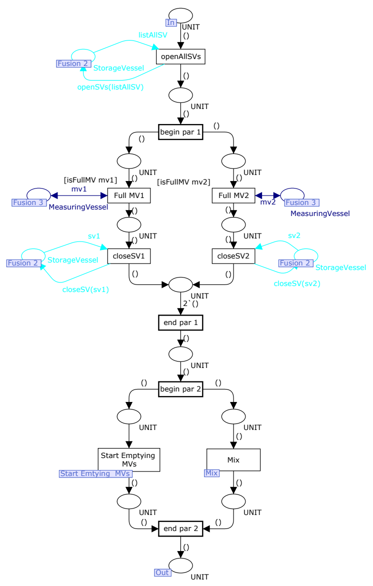


Fig. 15: CPN to represent the behavior of vessels (see Fig. 12)

4.4 Animation

We have developed a *SceneBeans* [7] animation to be associated with the created CPNs, through the *BRITNeY* animation tool [8]. A screen shot of the animation of reactor system is shown in Fig. 16.

In this animation we can find the elements of the problem domain, which is the reactor domain. We obtain this animation using the Scenebeans animator. On the left top side of the image we have commands accepted by the animation. On the left button side, we have the events produced by animation. These set of commands and events are use to do the interaction between the animation and the CPN models. For example, to animate the message "`openAllSVs`" in SD of Fig. 12, we invoke, in the corresponding transition, two commands of the animation: "`openSV1`" and "`openSV2`". The user can interact with the animation using the start button. The vessels on the top are the storage vessels. Each storage vessel has a corresponding measuring vessel. In the center we have the reactor vessel, with a mixer inside of it. On the button we have which transports the liquid to the unloading area.

This animation is intended to help us in validating the obtained CPNs in terms of their appropriateness to express in the user's domain language the requirements of a given system. Additionally we plan to use the animation to compare the two CPNs for the reactor system (Figs. 10 and 14) and to evaluate the performance of the rules to generate "good" CPNs. We would like to

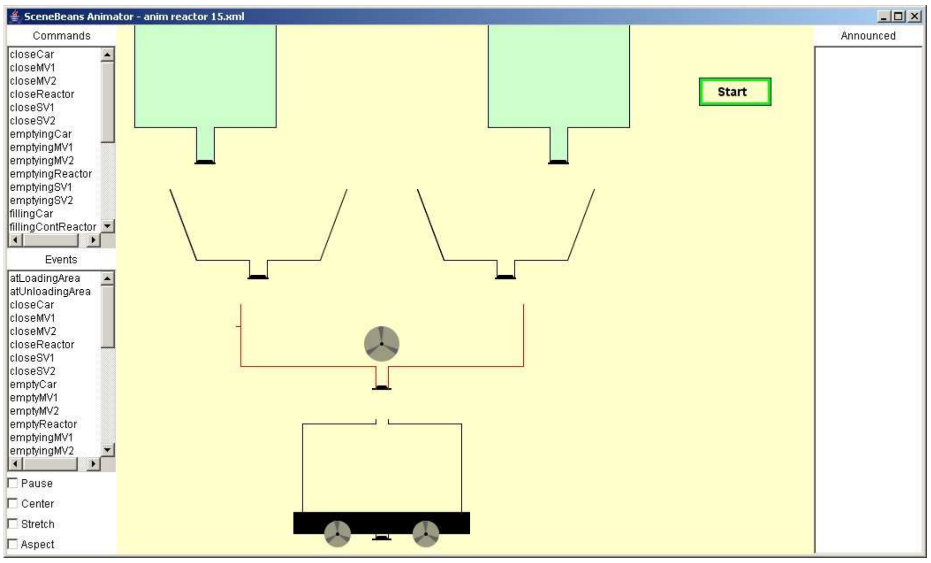


Fig. 16: The animation of the reactor system

observe if the animations controlled by both CPNs produce the same externally observable behavior, independently of their internal structures.

5 Related work

Transforming scenarios into state-based models (namely, sequential finite-state machines) has been the subject of many researchers. In fact, several approaches were already proposed to combine the usage of both scenarios and state-based models and, in this section, some of them are discussed. A major problem for obtaining state-based models from scenarios is the big computational complexity of the synthesis algorithms that does not allow the technique to scale up. Some additional obstacles include methodological issues, the definition of the level of detail in the scenarios to allow effective synthesis, to the problem of guaranteeing that the scenarios are representative of the users' intentions.

The majority of the approaches propose the usage of FSM (finite state machine). Krüger et al. suggest the usage of Message Sequence Charts (MSCs) for scenario based specifications of component behavior, especially during the requirements capture phase of the software process [9]. They discuss how to schematically derive statecharts from MSCs, in order to have a seamless development process. Harel proposes the usage of scenario based programming, through UML (Unified Modeling Language) [2] use cases and play in scenarios [10]. Harel's play in scenarios make it possible to go from a high level user friendly requirements capture method, via a rich language for describing message sequencing, to a full model of the system, and from there to the final implementation.

Whittle and Schumann propose an algorithm to automatically generate UML statecharts from a set of UML sequence diagrams [11]. The usage of this algorithm for a real application is also presented [12, 13], and the main conclusion is that it is possible to generate code mostly in an automatic way from scenario based specifications.

Hinchey et al. propose a round trip engineering approach, called R2D2C (Requirements to Design to Code), where designers write requirements as scenarios in constrained (domain specific) natural language [14]. Other notations are however also possible, including UML use cases. Based on the requirements, an equivalent formal model, using CSP, is derived, which is then used as a basis for code generation.

Uchitel and Kramer present an MSC language with semantics in terms of labeled transition systems and parallel composition [15]. The language integrates other languages based on the usage of high level MSCs and on the identification of component states. With their language, scenario specifications can be broken up into manageable parts using high level MCSs. These authors also present an algorithm that translates scenarios into a specification in the form of Finite Sequential Processes, which can be used for model checking and animation purposes.

The synthesis of Petri nets from scenario-based specification is less popular than the one that generates FSMs, because Petri nets represent a model of computation, where parallelism and concurrency of activities are "natural" characteristics. We next describe some of the approaches proposed to obtain Petri nets from a set of scenarios. In [16], the authors present a polynomial algorithm to decide if a scenario, specified as a Labelled Partial Order, is executable in a given place/transition Petri net. The algorithm preserves the given amount of concurrency and does not add causality. In case the scenario is indeed executable in the Petri net, the algorithm computes a process net that respects the concurrency expressed by the scenario. Although quite useful, this technique is not yet available for high-level Petri nets, such as Object-oriented Petri nets, Colored Petri nets (CPNs), or Reference nets.

In [17] an informal methodology to map Live Sequence Charts (LSCs) into CPNs is presented, for allowing properties of the system to be verified and analyzed.

The formal translation of Interaction Overview Diagrams (IODs) into PEPA nets is described in [18]. PEPA nets constitute a performance modeling language that consists of a restriction of Petri nets, where tokens are terms of a stochastic process algebra [19]. The translation is based on the idea that the structure given by the IOD corresponds to the high level net structure of the PEPA net, and the behavior described in the IOD nodes (sequence diagrams) can be translated onto PEPA terms. The translation allows a designer to formally analyze UML 2.0 models, using the tools for PEPA nets.

A formal semantics by means of Petri nets is presented in [20] for the majority of the concepts of sequence diagrams. This semantics allows the concurrent behavior of the diagrams to be modeled and subsequently analyzed. Moreover, the usage of CPNs permits an efficient structure for data types and control elements. In their approach they use places to represent the messages, instead transitions as we do.

This work is based on the preliminary results presented in [21], where the authors show how the behavior of animation prototypes results from the translation of SDs into CPNs. We extend their results by showing how to translate more types of operators in UML 2.0 SDs, namely by considering parallel constructors which result in CPNs with true concurrency (i.e. CPNs that are not just sequential machines).

6 Conclusions

In this paper we show a set of rules to transform SD into equivalent CPNs for animation proposes. In UML 2.0, SDs are quite expressive and this work explores the new constructors (in relation to UML 1.x) that allow several plain sequences to be combined in a unique SD. Thus the rules allow the generation of a CPN that covers several sequences of behaviors. This work is in progress so we plan to develop it further. First we plan to investigate all SD operators, namely the *neg* operator and evaluate if it can be useful for the software engineer. Second, we need to better tune the rules, to realize if they can be automated. In the future we would like to use a UML-based tool to draw the SD diagrams and apply automatically the rules to obtain a CPN for animation. Probably this automation requires a second set of rules that "optimizes" the CPN by eliminating redundant parts. In this work we only have a validation of transformations though the implementation, we plan to study the soundness and completeness of the approach.

Finally the usage of the rules in real-world projects is planned, since we believe that methods and tools for software engineers need to be evaluated by them in complex industrial projects.

References

1. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Brauer, W. and Gozenberg, G. and Salomaa edn. Volume Volume 1, Basic Concepts of Monographs in Theoretical Computer Science. Springer-Verlag (1997) ISBN: 3-540-60943-1.
2. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modelling Language. Addisson-Wesley (2003)
3. Adamski, M.: Direct Implementation of Petri Net Specification. In: 7th International Conference on Control Systems and Computer Science. (1987) 74–85
4. Machado, R.J., Fernandes, J.M., Proença, A.J.: Specification of Industrial Digital Controllers with Object-Oriented Petri Nets. In: IEEE International Symposium on Industrial Electronics (ISIE'97). Volume 1. (1997) 78–83

5. Fernandes, J.M., Pina, A.M., Proença, A.J.: Concurrent Execution of Petri Nets based on Agents. In: Encontro Nacional do Colégio de Engenharia Electrotécnica (ENCEE95), Lisbon Portugal, Ordem dos Engenheiros (1995) 83–9
6. Ribeiro, O.R., Fernandes, J.M., Pinto, L.F.: Model Checking Embedded Systems with PROMELA. In: 12th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2005), Greenbelt, MD, E.U.A., IEEE Computer Society Press (2005) 378–85
7. Pryce, N., Magee, J.: SceneBeans: A Component-Based Animation Framework for Java. (Online) http://www-dse.doc.ic.ac.uk/Software/SceneBeans/.
8. Westergaard, M., Lassen, K.B.: Building and Deploying Visualizations of Coloured Petri Net Models Using BRITNeY Animation and CPN Tools. In: Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools. (2005)
9. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In Rammig, F.J., ed.: Distributed and Parallel Embedded Systems, Kluwer Academic Publishers (1999) 61–71
10. Harel, D.: From play-in scenarios to code: An achievable dream. IEEE Computer **34**(1) (2001) 53–60 (Also, Proc. Fundamental Approaches to Software Engineering (FASE; invited paper), Lecture Notes in Computer Science, Vol. (Tom Maibaum, ed.), Springer-Verlag, March 2000, pp. 22-34.).
11. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: 22nd International Conf. on Software (ICSE), Limerick, Ireland (2000) 314–323
12. Whittle, J., Saboo, J., Kwan, R.: From scenarios to code: An air traffic control case study. In: ICSE'03. (2003) 490–497
13. Whittle, J., Kwan, R., Saboo, J.: From scenarios to code: An air traffic control case study. Software and Systems Modeling **4**(1) (2005) 71 – 93
14. Hinchey, M.G., Rash, J.L., Rouff, C.A.: A formal approach to requirements-based programming. ecbs **00** (2005) 339–345
15. Uchitel, S., Kramer, J.: A workbench for synthesising behaviour models from scenarios. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada, IEEE Computer Society (2001) 188–197
16. Juhs, G., Lorenz, R., Desel, J.: Can I Execute My Scenario in Your Net? In Ciardo, G., Darondeau, P., eds.: Applications and Theory of Petri Nets 2005: 26th International Conference (ICATPN 2005). Volume 3536 of LNCS., Miami, USA, Springer (2005) 289
17. Amorim, L., Maciel, P., Nogueira, M., Barreto, R., Tavares, E.: A methodology for mapping live sequence chart to coloured petri net. In: IEEE International Conference on Systems, Man and Cybernetics. Volume 4. (2005) 2999–3004
18. Kloul, J., Kuster-Filipe, J.: From interaction overview diagrams to pepa nets. In: Proceedings of the 4th Workshop on Process Algebras and Timed Activities (PASTA'05), Edinburgh (2005)
19. Gilmore, S., Hillston, J., Kloul, L., Ribaudo, M.: Pepa nets: a structured performance modelling formalism. Performance Evaluation **54**(2) (2003) 79–104
20. Eichner, C., Fleischhack, H., Meyer, R., Schrimpf, U., Stehno, C.: Compositional semantics for UML 2.0 sequence diagrams using Petri Nets. In Lecture Notes in Computer Science, Volume, J.., ed.: SDL 2005: Model Driven Systems Design: 12th International SDL Forum. Volume 3530., Grimstad, Norway (2005) 133–148
21. Machado, R.J., Lassen, K.B., Oliveira, S., Couto, M., Pinto, P.: Execution of UML Models with CPN Tools for Workflow Requirements Validation. In: Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools. (2005)