A METAMODELING APPROACH TO TRANSFORM UML 2.0 SEQUENCE DIAGRAMS TO PETRI NETS

Michel dos Santos Soares
Delft University of Technology
Delft, The Netherlands
email: m.dossantossoares@tudelft.nl

Jos Vrancken
Delft University of Technology
Delft, The Netherlands
email: j.l.m.vrancken@tudelft.nl

ABSTRACT
The paper presents transformations from UML 2.0 Sequence Diagrams to Time Petri nets with inhibitor arcs. The transformation is not restricted to messages calls and responses, but also to some of the new Sequence Diagrams operators. This model-to-model transformation is based on metamodels and is useful to improve semantics, as Petri nets can be executed by simulation and formally proved. The advantages of this multi-formalism approach are that different views are represented, complex systems development is done based on several levels of detail, and complexity is managed with abstraction and modularization.

KEY WORDS
Sequence Diagrams, Petri nets, Model Driven Engineering, Formal Methods, Model Transformations.

1 Introduction

Software intensive systems are complex systems that are difficult to model, design and analyze. In these systems, software interacts with other software, systems, devices, sensors, actuators and with people. Their complexity is increased due to the large number of elements and reliability factors, mainly when these systems involve human life. They must be decomposed into several smaller components in order to manage complexity and facilitate their implementation and verification. It is then necessary to use methods that are capable of addressing complexity and providing rigorous solutions during their development.

Formal methods are capable of offering these characteristics by allowing formal proofs of desirable behavior and simulations that may help in discovering design problems. Also, modular design and abstractions that are important to manage complexity are common to several formal methods. But there are some disadvantages that concern industry when considering the application of formal methods in practice. One common problem is the lack of skilled personnel. Although advocated that formal methods are based on simple mathematics [1], on average they are more complicated to learn and use than semi-formal methods. Another disadvantage is that in large, complex systems, requirements are difficult to gather and very volatile, which means that using increased efforts (time, money, personnel) to create formal specifications that are certainly going to change several times seems to be counter-productive.

A common approach is then to start system design with semi-formal methods, in a very abstract level, and then increase the degree of formalization by using model-driven transformations. Then, formal methods can be used later in the design cycle, mainly for critical subsystems. This solution seems to be natural, as initial efforts are not useless, due to the possibility of transformation between models, and also the benefits of formal methods are considered.

UML is considered by the Software Engineering community a semi-formal language. The language is a compilation of other object-oriented methods created during the 80’s and 90’s and emerged as the de facto standard to software specification. The new version launched in 2005, UML 2.0 [2], is a major revision that brings improvements. In spite of the precise syntactic aspects of UML notations, the language is still often criticized [3] [4] because of its complexity, the lack of well-defined semantics leading to subjective interpretation on models, and the lack of verifiability capabilities.

In order to try to improve UML semantics and apply UML in the design of critical systems, some approaches were proposed combining UML diagrams to a formal method such as B or Z. UML-B [5] is a precise and semantically well-defined profile created by the combination of UML and the B-method. The profile has been used in several real-time control systems in industry projects. UML + Z [6] is a framework for building, analyzing and refining models of software systems based on UML and the formal specification language Z.

In terms of translating from UML dynamic diagrams to Petri nets, several works were published in the past years. UML provides diagrams that are strong to represent structure, which is not well-represented with Petri nets. On the other hand, dynamic behavior is better defined with Petri nets than UML. In [7] UML dynamic diagrams are used for system modeling and performance evaluation is realized by using Stochastic Petri nets. In [8] an approach to translate UML statecharts to time Petri nets is presented, with the purpose of performing formal analysis by checking temporal constraints. In [9] Sequence Diagrams and some fragments are transformed to Coloured Petri nets and the semantics is enriched with Algebra. In [10] a set of rules to transform Sequence Diagrams into equivalent Coloured Petri nets for animation purposes using computer
tools is shown. In [11] a metamodeling approach is proposed to transform Sequence Diagrams to Petri nets, integrating early in the life cycle formal verifications. Few of these works have used explicit metamodeling or considered time constraints in the transformations, as proposed in this paper.

The context of the paper is a model-driven approach applied to complex systems. It proposes a model-to-model transformation, from UML 2.0 Sequence Diagrams, including some of the new operators, to Petri nets. The transformation is not represented considering just the fragments, but also the communication between objects, which is the main characteristic of Sequence Diagrams. The time constraints are formally represented with time intervals associated to Petri nets places. The dynamic internal behavior of communicating objects are specified with Petri nets. Also, it is considered the communication between objects in a Sequence Diagram given by Petri nets places. The Petri nets and Sequence Diagram metamodels are also presented.

2 Model Driven Engineering

In traditional Software Engineering, models are not used in all their potential. Normally they are the base to write code, or just used as a documentation effort, after the “real system” is ready. These documents are mainly based on semi-formal notations or informal diagrams, with some add of natural language. The most important artifact is the code, and once this is working, models, if they exist, are often discontinued or discarded. During development, changes in the code are not always implemented also in models. In practice, the result is that after few weeks of development, the code is different from the models, and even small changes are based on trying to understand several lines of code, a difficult, costly and error-prone task.

With the growing complexity of systems, several teams from diverse domains, increased necessity of maintenance and evolution for long periods, a new approach is needed. The challenge is then to improve models importance, even small changes are based on trying to understand several lines of code, and to transition from document-centric to model-centric. In this Model-Driven Engineering (MDE) [12] approach to system development, models are as important as they are in other engineering disciplines. One of the best known model-driven approaches is OMG’s MDA [13].

The early integration of a formal method in the system design process has many advantages, such as improved semantics, model execution by simulation and formal proofs, contributing to the early validation of requirements. Executable models, such as Petri nets, can complement semi-formal specifications, such as Sequence Diagrams. There are several tools that can be applied to simulate Petri nets [14]. Simulation can be compared to program execution: it can show presence of errors/bugs, but cannot guarantee that the model is correct. Formal verifications based on mathematical/logical proofs are then necessary. Some examples of techniques are reachability enumeration, model checking and invariant analysis.

In order to define an MDE process, two important techniques are metamodeling and model transformation [12]. Metamodeling allows defining precisely a class of models, defining a modeling language by specifying its abstract syntax, eventually along with its semantics. Also, it facilitates the mapping between concepts of both metamodels. In MDE, a particular view of a system can be captured by a model and each model is written in the language of its metamodel (figure 1). The system level is the “real world” and the model and metamodel levels are the “modeling world”. Model transformation allows to clearly define relationships between models and depends heavily on the metamodels of the related models. The metamodels used in this paper are given in sections 3 (Sequence Diagram) and 4 (Petri nets), and the model-driven transformations in section 5.

3 UML Sequence Diagram Metamodel

Sequence Diagram is a UML interaction diagram which emphasizes message exchanging between objects over their life time. These messages can be, for instance, call for operations or signals. A relevant part of the Sequence Diagram metamodel is shown in figure 2. The diagram is normally useful to represent use case descriptions. The new UML 2.0 version brings several new features, improving modeling capacity of previous versions. The major changes are related to fragments of Sequence Diagrams that can be reused, and are of a given type. Also, Sequence Diagrams can be part of another Sequence Diagram, allowing composition and decomposition. In this paper, synchronous and asynchronous messages and the operators ALT (and OPT), LOOP and PAR are transformed into Petri nets. Other operators are considered in future research.

The operator ALT (alternatives) provides the possibility of a choice of behavior in which at most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction. Only the messages associated to the true guard (if any) are executed. The optional operator OPT can be seen as an alternative with only one operand.

The LOOP operator may specify a range from the minimum to the maximum number of iterations. Cons-
Petri nets are composed of four basic elements: places, transitions, arcs and tokens. Due to their representation power, the elements can interpret several different roles. For instance, places normally represent conditions, status or operations. Transitions are generally used to represent starting or stopping of events, which occurs to change the status of places. Arcs are connections between places and transitions, and tokens can represent number of elements or the current availability of resources.

Petri nets have mechanisms to provide important aspects for the design of large and complex systems, such as abstraction, hierarchical design and modularization. Transitions and places can represent sub-nets, allowing modeling in several levels of detail. In addition, it is feasible to construct large models relating smaller Petri nets to each other. An example is the fusion of common places, representing the same resource in two different Petri nets. Place fusion is a simple and effective way to model communication between sub-nets [16].

Formally, a Petri net is a 5-tuple \( N = (P, T, I, O, M_0) \), where: \( P = \{P_1, P_2, ..., P_n\} \) is a finite set of places, \( T = \{t_1, t_2, ..., t_m\} \) is a finite set of transitions, \( Pre : (P \times T) \rightarrow \mathbb{N} \) is an input function that defines directed arcs from places to transitions, and \( \mathbb{N} \) is the set of nonnegative integers, \( Post : (P \times T) \rightarrow \mathbb{N} \) is an output function that defines directed arcs from transitions to places, and \( M_0 : P \rightarrow \mathbb{N} \) is the initial marking. A marking is an assignment of tokens to places and is fundamental for the analysis of Petri net models. If \( Pre(p, t) = 0 \), then there is no directed arc connecting \( p \) to \( t \). In a similar manner, if \( Post(p, t) = 0 \), there is no directed arc connecting \( t \) to \( p \). If \( Pre(p, t) = k \) (the same holds for \( Post(p, t) = k \)), then there exist \( k \) parallel arcs connecting place \( p \) to transition \( t \) (transition \( t \) to place \( p \)). Graphically, parallel arcs are represented by a simple arc with a number as label and representing the presence of multiple arcs (weight \( k \)).

The dynamic behavior (execution) of a Petri net is described by changing markings (from \( M \) to \( M' \)) when an event occurs, which is represented by the firing of an enabled transition. A transition \( t \) is enabled if each place \( P \) that has an input arc to \( t \) contains at least the number of tokens equal to the weight of the arcs connecting \( P \) to \( t \) \((M(p) \geq Pre(p, t) \) for any \( p \in P \)). The flow of tokens represent state changing. The firing of an enabled transition \( t \) removes from each input place \( p_i \) the number of tokens equal to the weight of the directed arc connecting \( p_i \) to \( t \), and deposits in each output place \( p_o \) the number of tokens equal to the weight of the directed arc connecting \( t \) to \( p_o \). Therefore, the firing of a transition yields a new marking \( M'(p) = M(p) - Pre(p, t) + Post(p, t) \) for any \( p \in P \).

In order to improve Petri nets applicability, several extensions to the basic theory were proposed. These extensions change the behavior or just increase modeling power. Two of these extensions are considered in this paper: the
association of time to places, and the representation of priorities and conditions using inhibitor arcs. The extensions are explained in the following subsections and the correspondent metamodel is shown in figure 3.

4.1 P-Time Petri nets

A p-time Petri net [17] is a pair $< N, Ip >$ where $N$ is a Petri net and $Ip$ is the application defined as $Ip : P \rightarrow (\mathbb{Q}^+ \cup 0) \times (\mathbb{Q}^+ \cup \infty)$.

For each place $p_i \in P$ is associated an interval $Ip_i = [\alpha_i, \beta_i]$ with $0 \leq \alpha_i \leq \beta_i$. The static interval $Ip_i$ represents the permanency duration of a token in $p_i$. Before the duration $\alpha_i$, the token in $p_i$ is in the non-available state. After $\alpha_i$ and before $\beta_i$, the token in $p_i$ is in the available state for transition firing. After $\beta_i$, the token in $p_i$ is again in the non-available state and cannot enable any transition anymore. This token is named “dead token”, and can be seen as a time constraint that was not respected. The dynamic evolution of a p-time Petri net model depends on the marking $M$ and on the time situation of the tokens (non-available, available, or dead). P-time Petri nets represent formally time constraints that are very important in real-time systems. This is an advantage over Sequence Diagrams, that only represent time with informal notations associated to the life of the object.

4.2 Inhibitor arc Petri nets

Inhibitor arcs are a mechanism to extend Petri nets in order to represent conditions and priorities. An inhibitor arc connects a place to a transition, and is represented by an arc terminated with a small circle (instead of an arrow in a classic input arc).

In a Petri net with inhibitor arcs [18], the firing of a transition depends not only on the transition being enabled, but also on the presence of an inhibitor arc as input to the transition. In the presence of an inhibitor arc, the transition cannot fire if one or more inhibitor arcs are incident to the transition, and at least one of them has as origin a place containing the number of tokens equal to the weight of the arc. No tokens are moved through an inhibitor arc when the transition fires.

The figure 4 shows an example. Transition t2 has priority over transition t1, which is represented by an inhibitor arc from place B into transition t1. As there is a token in place B, then transition t1 cannot fire, even with place A enabling the transition. In the situation in figure 4, although transitions t1 and t2 are both enabled, first transition t2 is fired.

Inhibitor arc Petri nets has the possibility of testing whether a place is empty in the current marking. This means that inhibitor arcs are well suited to model situations involving testing for a specific condition, as stated by ALT, OPT and LOOP operators in a Sequence Diagram.

5 Model Driven Transformations

This paper focus on model-to-model transformations. UML 2.0 Sequence Diagrams and some operators are transformed to Petri nets. It is assumed that the internal behavior of the communicating objects are also given by Petri nets. The main advantages of transforming from Sequence Diagrams to Petri nets are relative to the formal characteristics of the former.

Figure 5 shows the transformation from Sequence Diagram’s synchronous messages to p-time Petri nets. Synchronous messages are used when the sender waits for a response from the receiver to continue its own operations. This is common when the object sender depends on some operation of the object receiver to continue its execution. The message sent from one object to another is translated to a communication place that outcomes from the object sender and incomes to the receiver. In the example, objA sends the message msg1() to objB. Object objB process the message during a period of time: $dt$, represented in the Sequence Diagram, and $[0, dt]$ represented in the p-time Petri net. The interval represents the time when the operation starts (“0”) and ends (“dt”). After the processing, objB sends the response respMsg() back to object objA.

The asynchronous message is used when the sender continues execution after sending the message. This can
be useful when the sender wants to communicate through a signal with the receiver, but will not wait for a response. Figure 6 shows the transformation from asynchronous messages to Petri nets. The basic functionality is the same as the synchronous communication; the difference is that in asynchronous communications there is no need to wait for a response.

Figure 7 shows the transformation from the ALT operator to p-time Petri nets with inhibitor arcs. In the example, only 2 alternatives are shown for the ALT operator, but the transformation can be scalable to other alternatives with more transitions and inhibitor arcs. There must be a mechanism to decide for the firing of just one transition, as for instance, the weight of arcs or different time delays. Basically, the output alternatives for object ObjA are the communication places \([Alt1]msg1()\) or \([Alt2]msg2()\), based on satisfying condition Alt1 or Alt2. These alternatives are received respectively by objB-Alt1 or objB-Alt2. Due to its similarity with the ALT operator, it is possible to apply the same general translation scheme to operator OPT.

Figure 8 shows the transformation from the PAR operator to p-time Petri nets. Both operations initiated by messages msg1() and msg2() are performed in parallel in the object receiver. Object sender can only continue its own processing when both are finished.

Figure 9 shows the transformation from the LOOP operator to p-time Petri nets. For sake of simplicity, only loops with fixed number of iterations are presented. The number of iterations is controlled in the Petri net using an inhibitor arc, which stops the loop after “n” iterations. The loop may also be controlled by a constraint, such as the end of an operation in the object receiver. Then, a similar transformation can be made, by using a communication place indicating that the operation has ended as an inhibitor arc, instead of the number of iterations.

6 Rules Validation

The approach has been applied in the modeling of real-time complex systems in different domains. Initially, requirements are modeled with UML Use Case Diagrams that are later specified in more details using Sequence Diagrams.
Then, the Sequence Diagrams are translated into Petri nets, which are executed using computer simulators and proved using one or more verification methods. When necessary, corrections are made in the Sequence Diagrams and Class Diagrams. For instance, new methods or attributes may be created in a class. Time constraints, which cannot be verified with Sequence Diagrams, are well-represented with Petri nets and also simulated. The effort is useful to improve semantics of UML diagrams, to execute several Sequence Diagrams scenarios through Petri nets simulation, and to early verify models reliability.

7 Conclusion

The paper presented a model-driven approach to transform UML 2.0 Sequence Diagrams to Petri nets with time associated to places and with inhibitor arcs. The transformation is not restricted to synchronous and asynchronous messages, but also to the new Sequence Diagram operators Loop, Alt, Opt and Par. The transformation is useful to improve Sequence Diagrams semantics. For instance, time constraints are formally represented with p-time Petri nets. Another advantage is that, as a formal method, Petri nets may be simulated and formally proved, which is useful to discover problems in the early phases of system design. In future research, the approach will be applied to more case studies, and it will be investigated the transformation of other operators.

References