INTEGRATED TYPE ANALYZER AND STATE SPACE GENERATOR OF OBJECT-ORIENTED PETRI NETS

Bohuslav Křena, Tomáš Vojnar, Milan Češka

Faculty of Information Technology, Brno University of Technology,
Božetěchova 2, CZ-612 66 Brno, Czech Republic
E-mails: {krena,vojnar,ceska}@fit.vutbr.cz

Abstract— The paper deals with the problem of conjunct implementation of type analyzer and state space generator in the context of the (syntactically) weakly typed object-oriented Petri nets associated with the PNtalk language. The main principles of object-oriented Petri nets, their state spaces generation, and type analysis (i.e. deriving the types of tokens that may get into particular places of PNtalk-based models) are introduced here. The paper discusses advantages and disadvantages of the conjunct tool implementation.

Key Words— Concurrent systems; Discrete-event systems; Formal methods; Object modelling techniques; Parallelism; Petri-nets; System analysis.

Nomenclature

BT . . . Black Token
INT . . . Integer
JVM . . . Java Virtual Machine
OOPN . . . Object-Oriented Petri Nets

1 Introduction

The object-oriented Petri nets and the language and the tool called PNtalk (Janoušek, 1998) based on them are the merits of our research activities. The OOPNs are convenient for modelling, simulating, analyzing and verifying concurrent object-oriented software systems in a natural way. OOPNs features implemented in PNtalk involve all the key features of these systems as object-orientation concepts, communication by message sending, concurrency of events and processes, and synchronization.

The first prototype of a tool for simulating systems described by PNtalk was implemented several years ago (Češka et al., 1997). Currently, a new implementation of the simulator is being prepared. The new PNtalk engine will allow modellers to run OOPN-based prototypes in a truly distributed way (Češka et al., 2002). Moreover, some work has already been done also on state-spaces-based methods intended for formally analyzing and verifying properties of systems modelled by OOPNs (Češka et al., 2001; Vojnar, 2001).

PNtalk OOPNs—unlike many of the common dialects of high-level Petri nets (Jensen, 1992; Lakos and Keen, 1994; Mäkelä, 2001; Pet, 2002)—are not (syntactically) strongly typed which means that modellers do not have to explicitly declare the types (i.e. colour sets) of OOPN places, the types of the variables used in OOPN inscriptions, and so on. Reference nets used in the Renew tool (Kummer et al., 2001) (probably the closest dialect of high-level nets with respect to PNtalk OOPNs) allows modellers to assign types to none or all inscription variables and to some places. The typing information provided by the modellers is then used to statically look for some inconsistencies in a model. However, this is different from the approach discussed here which tries to automatically compute the types of tokens that can get to particular places of an OOPN. Weak typing of PNtalk is useful in the area of prototyping, however, as mentioned below, there are situations in which it may be useful to know at least something about the types associated with particular OOPN places.

Type analysis can be understood as a special means of debugging OOPN-based models: Modellers usually have some intuition about what types of tokens should get into particular OOPN places and if the results of a type analysis run on their model are different, there may be a fault in this model. The information derived from type analysis can also be used as a part of the documentation of a model. Moreover, as most Petri net dialects as well as various other formalisms seem to be strongly typed, the results obtained from type analysis can further be useful when translating PNtalk OOPNs into models described by some other modelling languages. The possibility of such a translation may be interesting from the point of view of re-using the tools already developed for working with various kinds of models. Finally, the results which may be obtained from type analysis may be used to optimize the internal representation of OOPNs and thus to increase the efficiency of running OOPN-based models and generating their state spaces.

In the following, we firstly present the basic principles of the considered OOPNs. Subsequently, we describe major properties of state space generation and type analysis in the context of OOPNs. Next, we deal with the main design issues of a software tool which will be able to manage type analysis and (possibly parallel) state space generation.

2 Object-Oriented Petri Nets

First in this section, we explain the main principles of the structure and behaviour of OOPNs (the notation of PNtalk is illustrated in Fig. 1). A deeper introduction to the OOPN formalism can be found, e.g., in (Češka et al., 1997) and the formal definition of OOPNs in (Janoušek, 1998; Vojnar, 2001). Next, we deal with the issue of state spaces of OOPNs and the process of their generation, and, at the end of this section, we introduce several approaches (two of them are applicable in our opinion) to type analysis in OOPNs (Křena and Vojnar, 2001).

2.1 The OOPN Formalism

The OOPNs associated with the PNtalk modelling language are characterized by a Smalltalk-based object-orientation enriched with concurrency and polymorphic transition execution which allows message sending, waiting for and accepting responses, creating new objects, and performing primitive computations (Janoušek, 1998).

The Structure of OOPNs

An *object-oriented Petri net* is defined on a collection of elements comprising constants, variables, places, transitions, object nets, method nets, synchronous ports, message selectors, classes, object identifiers, and method net instance identifiers. An OOPN has its initial class and initial object identifier as well.

A *class* of an OOPN is given by its object net, its sets of method nets and synchronous ports, and a set of message selectors corresponding to its methods and ports. Object nets describe what data particular objects encapsulate and what activity they exhibit on their own. Method nets specify how objects asynchronously respond to received messages. Synchronous ports allow to remotely test and change states of objects in an atomic way. Inheritance of PNtalk classes is defined via inheritance of places and transitions of object nets, together with inheritance of sets of method nets and synchronous ports.

Object nets consist of places and transitions. Each place has some (possibly empty) initial marking. Each transition has conditions and preconditions (i.e. inscribed testing and input arcs), a guard, an action, and postconditions (i.e. inscribed output arcs).

Method nets are similar to object nets but each of them has additionally a (possibly empty) set of parameter places and a precisely one return place. When a method is started, parameter tokens are put into the appropriate parameter places. On the other hand, the method may be terminated when a token appears in the return place. Method nets can access places of the appropriate object nets, which allows running methods to modify the states of the objects which they are running in.

Synchronous ports are special transitions that cannot be fired alone but only dynamically fused to some regular transitions. These transitions "activate" the ports from their guards via message sending. Every synchronous port embodies a set of conditions, preconditions, and postconditions over places of the appropriate object net, optionally a guard expression, and a possibly empty set of parameters.

The Dynamic Behaviour of OOPNs

A state of an OOPN can be encoded as a *marking*, which can be structured into a system of objects. Thus the dynamic behaviour of OOPNs corresponds to an evolution of a system of objects. An *object* of a certain class c is a system of net instances that contains exactly one instance of the object net of c and a set of currently running instances of method nets from c. Every *net instance* entails its identifier and a marking of its places and transitions. A *marking of a place* is a multiset of elements of the universe. A *transition marking* is a set of records about method net instances invoked from the appropriate transition.

For a given OOPN, its *initial marking* corresponds to a single, initially marked object net instance from the initial class. A change of a marking of an OOPN is the result of an occurrence of some *event*. Such an OOPN event is given by (1) its type, (2) the identifier of the net instance it takes place in, (3) the transition it is statically represented by, and (4) the binding tree containing the bindings of the variables used on the level of the involved transition as well as within all the synchronous ports (possibly indirectly) activated from that transition. There are four types of events: `A` – an atomic action involving trivial computations only, `N` – a new object instantiation via the message `new`, `F` – an instantiation of a Petri-net described method, and `J` – terminating a method net instance.

A Simple Example of an OOPN

Let us now return to the example of an OOPN from Fig. 1. This OOPN consists of two classes—`Stack` and `Application`. The class `Stack` models the common abstract data structure stack while the class `Application` illustrates using of `Stack`.

The class `Stack` consists of an object net whose instances are intended to store the current local state of stack objects, two method nets that can be used to change this state, and a synchronous port which can be used to test the state of the stack. The place `top` always contains the index of the top element of the appropriate stack

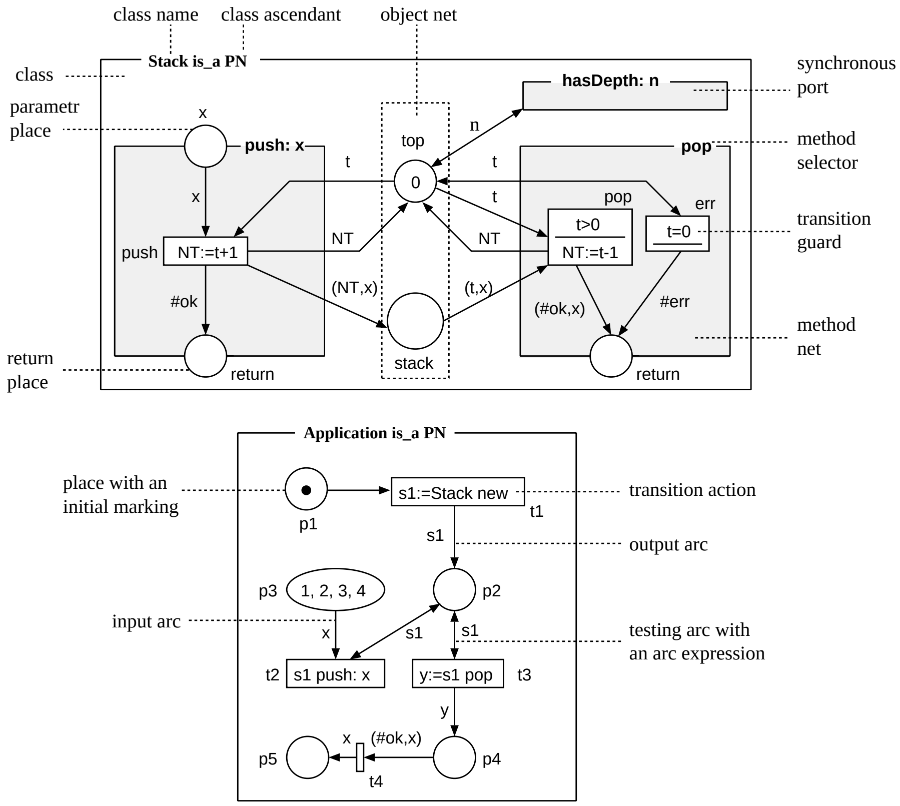

Figure 1. Two simple classes demonstrating the notion of the OOPN formalism.

object and the place `stack` contains a set of indexed elements currently stored in the stack. The method `push:` of `Stack` is used for pushing elements into a stack while the `pop` method is used for retrieving elements from the stack. The synchronous port `hasDepth:` can be used to test the depth of a stack from a remote object.

The class `Application` consists of an object net only. When an object of this class is created, its object net instance creates a new object of the class `Stack` by the action of the transition `t1` and stores it into the place `p2`. Tokens from the place `p3` are then pushed into the stack by the transition `t2` and popped from the stack by the transition `t3`. The transition `t4` moves tokens corresponding to valid objects obtained from the stack into the place `p5` while it leaves error tokens in the place `p4`.

2.2 State Spaces of OOPNs

State space can in general be defined as quadruple consisting of a set of states, a set of structural transitions, a set of semantic transitions (i.e. links between states and structural transitions), and an initial state. Sometimes it is useful to deal with a set of initial states instead of a single initial state. In the context of OOPNs, we can define (full) state spaces in the following way (Vojnar, 1999). For a given OOPN, states will correspond to reachable markings and structural transitions to events that can become enabled. Semantic transitions will be defined as triples consisting of a reachable marking M, an event E enabled in M, and the marking we obtain when we fire the event E in the source marking M. Finally, the initial state will be the initial marking, of course. It is necessary to pay a careful attention to the fact that particular states of OOPNs contain identifiers of dynamically arising and disappearing instances. The instances that exist in a given state of a system have to be uniquely identified to make it possible to refer to the local states of the instances and to express references among them. However, the presence of instance identifiers in states can significantly contribute to the state space explosion problem. This is due to the possibility of unnecessary generating many redundant states differing only in the names of the involved instances even if the names cannot influence the future behaviour of the system being examined in any way. We denote this phenomenon as the naming problem. The naming problem can be solved either by restricting the semantic of the applied modelling language or by examining the appropriate state spaces via their suitably reduced representation (we prefer the laster). Two methods for solving the naming problem were proposed in (Vojnar, 1999): (1) using sophisticated naming rules for assigning identifiers to newly arising instances and (2) the so-called name abstraction as a transparent application of the symmetry method for reducing state spaces.

2.3 Type Analysis in OOPNs

The main goal of type analysis is to derive the types of tokens that may get into particular places of OOPNs models. There can be identified two extreme approaches. Firstly, we may immediately assign the union of all the types which appear in the given OOPN to every OOPN place. On the other hand, we may explicitly generate and explore the full state space of the OOPN (if the state space is not infinite), which allows us to precisely find out what types of tokens may get into particular places. The first approach is extremely inaccurate while the second one quite expensive (Valmari, 1998).

Two more convenient approaches to type analysis in OOPNs are the so-called *static* and *dynamic type analysis* (Křena and Vojnar, 2001). The difference between static and dynamic type analysis consists in working with object instances: the dynamic one distinguishes different object instances while the static one does not. In order to decrease time and space requirements of type analysis and to ensure its termination, the static and dynamic OOPN type analysis methods suitably approximate the behaviour of OOPNs. The price which we pay for this is the fact the sets of the types of tokens about which we are informed that they can get into particular places can be bigger than in reality. However, this is a safe approximation and the obtained results can still be quite useful.

The static as well as the dynamic type analysis methods work with symbolically represented markings and events which do not contain concrete constants and unambiguous identifiers of Petri net-based objects but the corresponding types only. Accordingly, we do not perform classical trivial computations in guards of transitions and ports or in actions of transitions. Instead, we only derive the types of the involved variables. Next, as we work with sets of types symbolically representing multisets of tokens, the multiplicities of elements in initial markings and arc expressions are ignored. Moreover, we do not remove types symbolically representing some tokens once they get into a place.

Some of the principles mentioned above are illustrated in Fig. 2 where the class `Application` belongs to the model from Fig. 1. In Part b of the figure, we can see the symbolic "type" marking corresponding to the original initial marking (Part a). When we symbolically fire the transition `t1` from the initial symbolic marking, we obtain the symbolic marking shown in Part c of Fig. 2. In Part d of the figure there is shown the symbolic marking that can be obtained by repeatedly symbolically firing all the involved transitions and that cannot be modified any more and so it represents the result of the type analysis process.

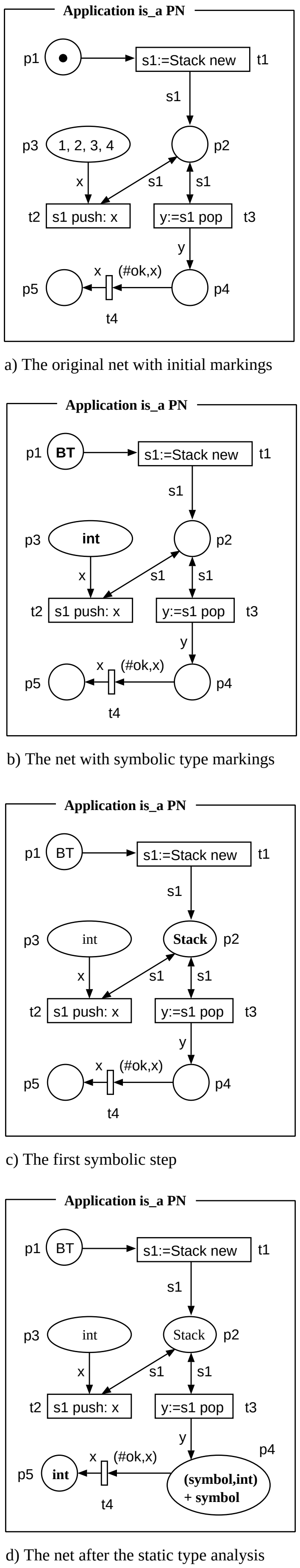

Figure 2. An illustration of the process of type analysis.

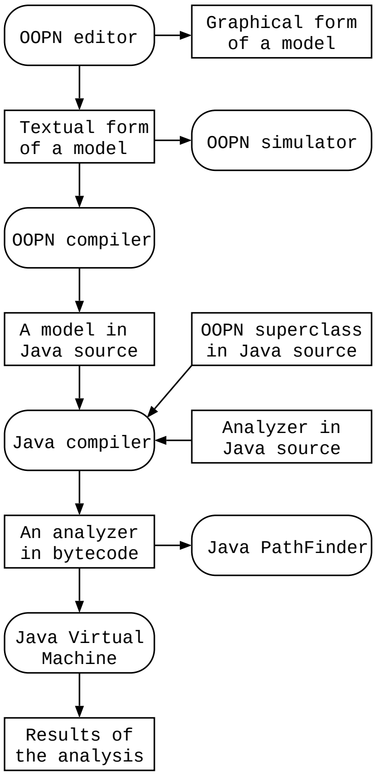

Figure 3. Structure of documents and program fragments of integrated OOPN analyzer.

3 Integrated OOPN Type Analyzer and OOPN State Space Generator

We decided to implement the OOPNs type analyzer and OOPNs state space generator together because some parts of software code seem to be similar. This approach is possible because state space generator (previously implemented in Prolog) also belongs into a field of our interests.

3.1 Process of Type Analysis and State Space Generation

The structure of software components and documents during type analysis and state space generation can be viewed in Fig. 3. Initially, we need an OOPN model of a system under investigation. A modeller can create an OOPN model in some OOPN editor or he/she can describe the system directly in textual form. We can simulate the model or translate it using OOPN compiler into some other formalism. Output language of OOPN compiler can be different in dependance on a purpose. There exists an OOPN compiler into internal representation of OOPNs in Prolog language. Modification of this compiler for translation OOPN models into Java source code seems to be quite easy.

Common properties of all OOPN classes will be described in a special Java class (in Fig. 3, called as OOPN superclass) which will be an ascendant of OOPN classes. Java code of OOPNs type analyzer and OOPNs state space generator will be placed into another Java class (we denote it as "analyzer" in Fig. 3). All these Java classes (type analyzer and state space generator, OOPN superclass, classes derived from a model) can be compiled by a standard Java compiler. The result of standard Java compilation is an analyzer (interpretable by standard JVM) specific for given OOPNs. Java byte-code of the model classes (maybe with some additional support) can be verified by Java PathFinder (Brat et al., 2000).

3.2 Advantages and Disadvantages of Presented Approach

Advantages of this approach are:

- We can use object orientation provided by Java and therefore we do not need to deal with classes and objects, inheritance, encapsulation, polymorphism, garbage collection, etc. in our own.
- We can use some parts of existing OOPN compiler from OOPNs textual form into representation in Prolog.
- It is also possible to verify Java bytecode (e.g. deadlock, assertions) by Java PathFinder.
- The tool will be portable because it uses Java programming language only.
- The tool can be modified for parallel running.

This approach has also some disadvantages:

- We must deal with Petri Net concepts (e.g. transition firing) in our own control, however, we must deal with this issue in most of possible approaches.
- We are limited by object orientation in Java language (e.g. we cannot change features of garbage collection in an easy way).

We suppose that the advantages of this approach are much more important than the considered disadvantages.

Parallelization of the tool

We are concentrated on parallelization of state space generator because it is a much more complex task than the type analysis. We consider parallelization based on systems with shared memory only because state space generation requires a huge amount of memory (parallelization based on systems with distributed memory is also possible but it is not so convenient and efficient).

4 Conclusion

In the paper, we have introduced OOPNs formalism, state spaces of OOPNs, and the type analysis. We have dealt with these issues previously and therefore only a brief introduction is offered here. The core of the paper has presented a discussion about implementation of integrated type analyzer and state space generator of OOPNs. Type analysis can be used to debug models, to improve their documentation, to make export to other languages easier, and for optimizations. Generated state spaces can be used to analyze and to verify models based on OOPNs. We have also discussed major advantages and disadvantages of the presented approach. In the near future, the described tool will be implemented and described in greater detail within Křena's doctoral thesis.

Acknowledgements

This work has been supported by the Ministry of Education of the Czech Republic within the project FR1948/2002/G1 "Formal methods of analysis and verification in object-oriented Petri nets". It was also supported by the Grant Agency of the Czech Republic under the contract 102/00/1017 "Modelling, Verifying, and Prototyping Distributed Applications Using Petri Nets".

References

Brat, G., Havelund, K., Park, S. and Visser, W. (2000). Java PathFinder – Second Generation of a Java Model Checker, *Workshop on Advances in Verification*.

Češka, M., Janoušek, V. and Vojnar, T. (1997). PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling, *in* F. Pichler and R. Moreno-Díaz (eds), *Computer Aided Systems Theory – EUROCAST'97*, Vol. 1333 of *Lecture Notes in Computer Science*, Springer-Verlag, Las Palmas de Gran Canaria, Spain, pp. 591–610.

Češka, M., Janoušek, V. and Vojnar, T. (2001). Generating and Using State Spaces of Object-Oriented Petri Nets, *International Journal of Computer Systems Science and Engineering* **16**(3): 183–193. Special issue: International Workshop on Software Engineering and Petri Nets (SEPN), editorial M. Pezzé and S.M. Shatz.

Češka, M., Janoušek, V. and Vojnar, T. (2002). Modelling, Prototyping, and Verifying Concurrent and Distributed Applications Using Object-Oriented Petri Nets, *Proceedings of the The 12th International WOSC Congress And the 4th IIGSS Workshop*. Accepted for publication.

Janoušek, V. (1998). *Modelling Objects by Petri Nets*, PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering and Computer Science, Technical University of Brno, Czech Republic. (In Czech).

Jensen, K. (1992). *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 1: Basic Concepts*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag.

Kummer, O., Wienberg, F. and Duvigneau, M. (2001). *Renew – User Guide (release 1.5)*, Department for Informatics, University of Hamburg, Hamburg, Germany.

Křena, B. and Vojnar, T. (2001). Type Analysis in Object-Oriented Petri Nets, *in* J. Zendulka (ed.), *Proceedings of 4th International Conference on Information System Modelling – ISM'01*, MARQ Ostrava, Hradec nad Moravicí, Czech Republic, pp. 173–180.

Lakos, C. and Keen, C. (1994). LOOPN++: A New Language for Object-Oriented Petri Nets, *Technical Report R94-4*, Department of Computer Science, University of Tasmania, Hobart, Tasmania 7001.

Mäkelä, M. (2001). Maria, Modular Reachability Analyzer for Algebraic System Nets, The user's guide for Maria 1.0.1, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland.

Pet (2002). The Petri Nets World. An online database of Petri net-related conferences, mailing lists, bibliographies, tool databases, newsletters, research groups, etc. URL: `http://www.daimi.au.dk/PetriNets/`.

Valmari, A. (1998). The State Explosion Problem, *in* W. Reisig and G. Rozenberg (eds), *Lectures on Petri Nets I: Basic Models*, Vol. 1491 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 429–528.

Vojnar, T. (1999). Towards Using State Spaces of Object-Oriented Petri Nets, *in* J. Štefan (ed.), *Proceedings of 33rd Spring International Conference on Modelling and Simulation of Systems MOSIS '99, Vol.1*, MARQ Ostrava, Rožnov pod Radhoštěm, Czech Republic, pp. 141–148.

Vojnar, T. (2001). *Towards Formal Analysis and Verification over State Spaces of Object-Oriented Petri Nets*, PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering and Computer Science, Brno University of Technology, Czech Republic.