Modeling Deliberative Agents using Object Oriented Petri Nets

Radek Kočí, Zdeněk Mazal, František Zbořil, and Vladimír Janoušek
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, Brno 61266, Czech Republic
{koci, mazal, zborilf, janousek}@fit.vutbr.cz

Abstract

Multi-agent systems are one of the prospect approaches to simplification of complex system development. They introduce a natural abstraction layer on top of autonomous actors. Since multi-agent systems are strongly parallel the appropriate paradigms or tools have to be used. This article will demonstrate the application of our developed kind of high-level Petri nets to the deliberative agent modeling and simulation. This formalism is called Object Oriented Petri Nets (OOPN). It introduces a powerful means for description of concurrency and a high level of system dynamism. Moreover, OOPNs are a part of a framework intended for simulation based design featuring model continuity. It means that a model developed incrementally in a simulated environment can become a part of target applications.

1. Introduction

There are three basic approaches to design artificial agent architectures. The key difference is whether the agent is endowed with a symbolic representation of the world it lives in or not. The agents that do not have any such representation are called reactive agents, whilst the other ones are called deliberative. The third possibility is to combine both approaches and create a layered architecture, such agents are called hybrid. All the three approaches have their advantages and disadvantages, which are discussed elsewhere [8].

Agent and multi-agent systems are inherently parallel – there are two main sources of parallelism. First of them is the parallelism in the whole system, i.e. the interaction among agents, concerning these autonomous units affect simultaneously their environment. The second source can be found inside the agent as most of the non-trivial agents are equipped with plenty of sensors and actuators, so they have to perform several tasks in parallel.

In this article we present our approach to modeling deliberative agents using the formalism of Object Oriented Petri Nets (OOPN). The motivation of our work is that Petri nets seem to be a suitable means to model agent systems. Their main advantages are pure description of concurrency, good comprehensibility and visual representation of the models combined with strong mathematical background, allowing automatical analysis and checking of model properties. There have been some works done in the area, for example [9] or [14]. These studies were implemented in the Design CPN tool, a well known coloured Petri nets (CPN) software. However, the CPN lacks one property, which we find essential for agent systems – dynamics.

OOPN and the associated PNtalk system were designed as an original attempt to bring high-level Petri nets closer to programming languages [4]. The project started as a consistent combination of high-level Petri nets and objects in Smalltalk. The main aspect of using OOPN and PNtalk for deliberative agent modeling is that it introduces a powerful means for description of concurrency and a high level of system dynamism. The other important advantage concerns a possibility to use OOPN as a part of Smalltalk environment in different scenarios. The model can be incorporated with Smalltalk program, so that it is connected to real environment [15]. Thus, the model can describe behavior of agent at the abstract layer but it also can define a real agent or its part. This property allows for safer development of agent-oriented software – thanks to a formal base of used paradigms we are able to check correctness of models and, consequently, these models are to be a part of developed software without needs for the model implementation in some programming language.

We will demonstrate principles of OOPN modeling on a simplified version of a classical example – Cleaner world. The article is structured as follows. Firstly, we will briefly describe the basis of OOPN. Then we will outline a deliberative agent and an idea about the Cleaner world example. Finally, we will show the model of Cleaner world in the OOPN formalism.

2. Object Oriented Petri Nets

General ideas about Petri nets have been introduced by C. A. Petri in early sixties. Longtime experiences have confirmed that Petri nets are powerful graphical tools having a mathematical foundation suitable for describing systems and their processes that can be termed as asynchronous, parallel, and distributed. Nevertheless, Petri nets lack a possibility of hierarchical description of models so it is hard to use them for complex systems modeling. To avoid this disadvantage, several extensions of Petri nets were designed. These Petri nets are called High-level Petri nets (HLPN). One of the HLPNs is the formalism of Object Oriented Petri Nets (OOPN) [4, 5] which has been developed at our faculty.

2.1. Basics of OOPN paradigm

OOPNs cover advantages of both kinds of used paradigms – Petri nets allowing developer to describe properties of the modeled system in a proper formal way and the object-orientation bringing structure and a possibility of net instantiation into models based on Petri nets. Object-orientation of the OOPN formalism is based on the well-known, class-based approach in Smalltalk-like style. It means that all objects are instances of classes, every computation is realized by message sending, and variables can contain references to objects. A class defines behavior of its instances as a set of methods (they specify reactions to received messages). A class is defined incrementally, as a subclass of some existing class.

OOPNs enrich the classical kind of object orientation by concurrency. Each object is an active server which offers reentrant services to other objects. It can also perform its own autonomous activity. Services as well as the independent activity are described by means of high-level Petri nets – services by method nets and object activity by object net.

The object net is instantiated immediately the new object is created. When some other object requests the object service, the appropriate method net instance is created and its execution starts. The net consists of places and transitions, tokens in the nets are references to objects. A transition has a guard restricting its firability and an action to be performed whenever the transition is fired. Arcs between places and transitions are inscribed by multiset expressions specifying multisets of tokens to be moved from/to certain places by the arcs associated with a transition being fired. The only way how to perform an object service (method) is to send a message to appropriate object from a transition action. A transition can be fired more times at the same time for various bindings (e.g., if a transition is conditioned by a place having two tokens – the transition can be fired twice). So the concurrency is not restricted to nets invocations, but the finest grains of concurrency in OOPN are transitions themselves.

The method net contains a special place named `return` providing a return action of the method. If some object is placed into the `return` place, it indicates the method invocation is able to finish. Once the calling transition claims this object then the method invocation is finished. To each argument of a method there is a corresponding place having the same name as an argument. These places are used for passing data (objects references) between calling transition and the method net.

Apart from method nets, classes can also define special methods called synchronous ports that allow for synchronous interactions of objects. This form of communication (together with execution of the appropriate transition and synchronous port) is possible when the calling transition (which calls a synchronous port from its guard) and the called synchronous port are executable simultaneously.

Every token points to an object, including numbers, strings, etc. An object can be either primitive (such as a number or a string – defined by Smalltalk), or non-primitive (defined by Petri nets). The way how transitions are executed depends on transition actions. A message that is sent to a primitive object is evaluated atomically (thus the transition is also executed as a single event), contrary to a message that is sent to a non-primitive object. In the latter case, the input part of the transition is performed and, at the same time, the transition sends the message. Then it waits for the result. When the result is available, the output part of the transition can be performed. In the case of the transition guard, the message sending has to be evaluated atomically. Thus, the message sending in a guard is restricted only to primitive objects, or to non-primitive objects with appropriate synchronous ports.

2.2. Interoperability

The formalism of OOPN forms a base of the PNtalk system. PNtalk provides a framework for simulation based system development. It means that the developer designs the system using formal models (OOPN) with which he is able to check a correctness of a modeled system by either formal verification or simulation methods. Moreover, our research assumes, that models can serve as a part of target application [7]. This implies, that there has to be a way of cooperation between OOPN and objects of different paradigms. So far, we have implemented an interoperability of OOPN and Smalltalk objects, so it is possible to transparently access OOPN objects from Smalltalk and vice versa. It is important for the reason of models deployment in real software applications.

The current PNtalk implementation allows for higher level of dynamism and higher level of control over the PNtalk features. PNtalk classes are special objects

which can be built incrementally during run time (like in Smalltalk). PNtalk classes and objects are implemented by their meta-objects in the PNtalk system architecture [5, 6]. Since these meta-objects can be manipulated programatically, this architecture allows for modifications of PNtalk classes as well as instances and their components (places and transitions) during run-time. Along with an interoperability it constitutes a basic means for the future research in the field of agent system design.

2.3. Illustration of OOPN

An example illustrating the OOPN formalism is shown in Figure 1. This example was demonstrated in [7] – we will show only small part of it. The fragment of the class `Conference` in Figure 1 has object net containing one place (`authors`), one method net (`addAuthorNamed:`), and one synchronous port (`author:`).

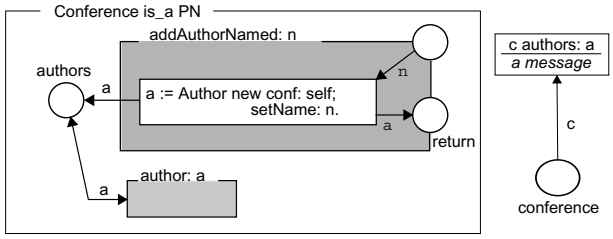

Figure 1. The OOPN example.

The method `addAuthorNamed:` serves for creating new author as an instance of the class `Author` and adding this object into the conference system (place `authors`). The synchronous port `author:` allows for testing whether an author passed by a formal argument `a` is added to a conference system.

Moreover, the synchronous ports can play another role. The transition testing for its firability and possible bindings has two basic steps. Firstly, the conditions (arcs) are resolved (and the variables are bound) and, secondly, the guard is evaluated. The guard can call synchronous ports. The transition can be fired only if all the called synchronous ports are able to fire simultaneously. If some transition requires the synchronous port call without bound arguments, the possible bindings of synchronous port arguments are resolved. For instance, if a transition calls the synchronous port `authors:` with unbound variable *a* (see the right side in Figure 1), then the synchronous port resolves (makes bindings for) all the registered authors – the transition will be enabled to fire for several different bindings (i.e. authors in our case).

3. Deliberative agents

The limit of reactive agents is their inability to plan next actions, which is caused by the fact that they do not have any representation of the world they are situated in. That means one cannot assign them tasks like *'move to place XY and there do action YZ'* because they simply do not know, where or what *XY* is. That is why we find the deliberative agents more perspective, although more difficult to create. Some of the most known deliberative architectures are Shoham's AOP (Agent Oriented Programming) [13] and the arguably most successful reification of the paradigm – the BDI architecture, based on the work of philosopher Bratman [1].

The BDI architecture has theoretical background – mainly the Rao and Georgeff's BDI CTL multimodal logics [12] as well as a lot of implementations: PRS [3], dMars [2], Jadex [10] etc. However there is a large gap between the theory and implementing systems. That is why many attempts to formalize the implementing architectures were undertaken, e.g., the AgentSpeak(L)[11] language or a formal description of the dMars system [2]. These specifications are again based on different kinds of logics, which means they are rather abstract, not very visual nor directly executable.

Our aim is to create a fully functional model of a complete BDI system using the OOPN formalism, which will allow us to experiment with different options and development choices in the architecture. In the following section we will illustrate some of the aspects of this problematics.

The BDI agent has typically several important components – beliefs about the world, plans, events, goals, intentions and the interpreter. In our case study, we will deal mainly with beliefs, plans and events.

4. A Model of the Cleaner World

As an example, we have selected the Cleaner world, a commonly used benchmark toy-application. Basic idea of the cleaner world is that an autonomous cleaning robot has the task to clean up dirt in some environment. A variant of such world has been used in article describing AgentSpeak(L), formal description of dMars and a more complex variant has been implemented in the Jadex system.

In the following text, due to limited length of the article, we use a very simple variant, which allows us to demonstrate the basic ideas. The world is defined by a discrete 2-D grid, some fields contain waste, which needs to be cleaned; the waste appears randomly. Apart from the waste, there are bins and an agent in the grid. The agent is able to perform these three atomic external actions: pick waste, move to an adjacent field and drop waste. It is able to carry one piece of waste at a time. The situation can look

like the one on the Figure 2. The grid contains a bin at position `(3,2)`, two pieces of waste at positions `(4,4)` and `(6,5)`, and an agent at position `(7,6)`.

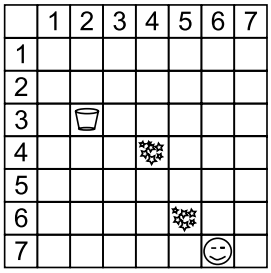

Figure 2. The Cleaner World example.

4.1. Agent's representation of the World

In deliberative agent architectures, the usual way to represent the state of the world is to use some kind of logic, e.g., first order logic. The representation of our example could look like this:

```
wastePosition(5,6).
wastePosition(4,4).
binPosition(2,3).
agentPosition(6,7).
carryingWaste.
```

Following the Prolog–like syntax, the semantics of agent's actions can be than transformed into four rules, manipulating the database with assert/retract clauses: `moveToWaste()`, `moveToBin()`, `pickWaste()`, and `dropWaste()`.

In OOPN, the most straightforward way to represent such facts is to create a place for each type of predicate, as it is shown in Figure 3. The tokens represent values of single predicates (in our example the structure denoted `(x,y)` represents a list and the symbol `#e` represents a black token without any specific value).

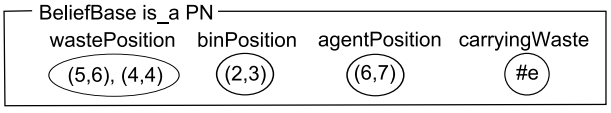

Figure 3. Agent's representation of the world in OOPN.

The actions can also be transformed into OOPN in a pretty straightforward way as transition with appropriate guards. The whole situation is shown in Figure 4.

Note that there are two places – `carryingWaste` and `notCaryingWaste` for one predicate, which is caused

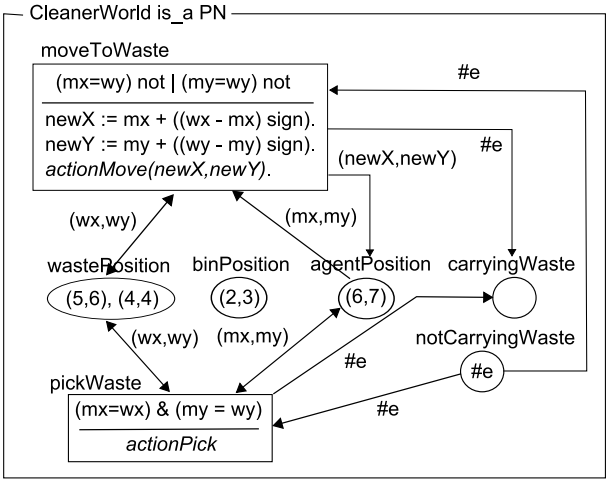

Figure 4. Representation of the Cleaner World in OOPN.

by the fact, that the current version of OOPN doesn't support inhibitor arcs. This property is not a problem from the theoretical point of view, as it doesn't affect the expressive power of the language, but it makes it more difficult to model some aspects (particularly a negation of a transition condition), which causes the models to get larger. That is why this property will be added into the language, so sometimes we skip the implementation details regarding this fact, as they can be translated into models runnable in the current version of PNtalk.

The model in Figure 4 can already be simulated. However, its behavior is non-deterministic when there appear more than one piece of waste or there is more than one bin in the system. That is why the next step is to introduce planning into the system. There are three main abstract concepts (classes) in our model – `Plans`, `Events` and `Agent`.

4.2. Plans

Plans represent agent's procedural knowledge how to achieve its goals. Our agent doesn't do any STRIPS–like first order planning, it only uses the plans prepared by the user (so called second order planning, common in BDI agent architectures). The concept of plans is modeled using a class hierarchy. The abstract plan provides means for plan identification (`plan-id`), plan lifecycle and the necessary reference to the agent, which allows the plans to test agent's belief base and ask it to perform the atomic external actions. The plan lifecycle defines states in which a plan can be – currently the states `ready`, `active`, `suspended`, `succeeded` and `failed` are supported. The change of the plan state is in most cases connected

with events (we will describe this further on in more detail).

The abstract plan has two important places in its object net – `start` and `finish`. These places are not connected, which means that the abstract plan cannot be run. Typically the programmer of the application inherits from this abstract class and adds the transitions and places to suit the concrete problem. In the Cleaner World model, the agent has three concrete plans – `CleanWastePlan`, `MoveToBinPlan` and `MoveToWastePlan`. A part of `CleanWastePlan` can be seen in Figure 5. The transitions perform some of the agent's atomic actions or they represent a subgoal, which can be achieved by invoking sub-plans (by posting internal events). Places represent the state of the plan. When a plan is created, a token is put into the place `start`. The transitions usually return tokens `#success` or `#failure`, which represent the result of their action and help to determine the selection of next transitions in the plan.

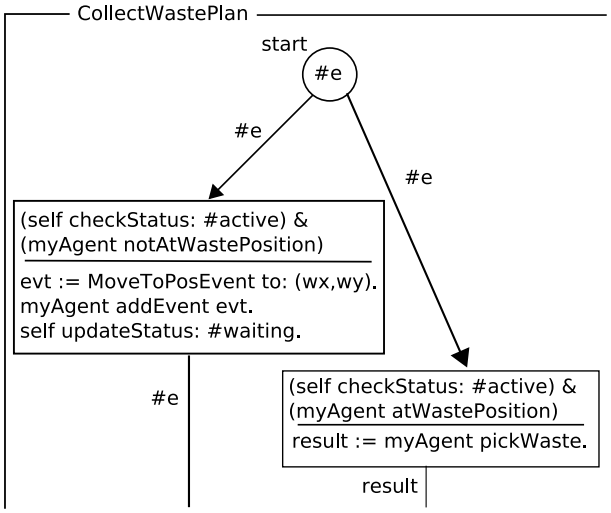

Figure 5. A small part of the CleanWastePlan, some details were cut out

4.3. Events

An abstract base class `Event` represents the second important concept in the system. Its subclasses provide abstraction for all the changes inside and outside the agent that are relevant. According to the place of origin of the event we distinguish external and internal events.

External events are generated by a transition, which models agent's sensors. In our example four basic types of external events can occur – appearing and disappearing of a piece of waste or a bin. These events are always application specific, so they must be created by the programmer of the application. They can carry arbitrary number of parameters (like the position of a new bin etc.). Internal events are generated inside the plans. They include requests on sub-plan invocation, notification on successful termination or failure of a plan etc.

All the events are put into a place called `EventQueue` located inside the agent class object net (see Figure 6), which provides means for scheduling the plans. Every plan has places and corresponding predicates, which can contain triggering events. In case there is an event in the event queue that matches one of these predicates, the plan's status is updated (e.g. it is set to *ready* when an event denoting a sub-plan has succeeded) or a new plan is created from the available templates.

The mechanism of event matching is shown in a guard of the first transition in Figure 6. For each plan its trigerring events are bound to variable `te` and tested whether satisfy conditions of the event or not. If such a plan is found, the transition is fired for this plan.

4.4. Agent

The main class of our example is the class `CleanerAgent`. It inherits from an abstract class `Agent` (a small part of its object net is shown in Figure 6). The class `Agent` provides means for management of the plans and events described above and the agent's lifecycle (interpretation of the plans, initialization, cleanup, etc.). We have already mentioned the place `EventQueue`, which is used to store all the events that appear in the system. Another important place in the object net is `PlanTemplates`, which stores samples of all the plans that the agent can use. Plans that were instantiated and concretized are gathered in a place called `PlanInstances`.

An object called `PlanScheduler` selects one of the plans that are in the state `ready` from the place `PlanInstances` (at present the scheduler doesn't do any optimizations, it just makes sure that only one plan is executed until it is finished or suspended). The state of the selected plan is set to `active`. This plan then performs one step (i.e. fires a transition in its object net, as described above).

The places `PlanTemplates` and `PlanInstances` are connected with the event queue by transitions, which allow changing the plan status or creating a new plan instance from template whenever an appropriate event occurs.

At current version we are not using the concept of intentions to manage the plans, although we plan to incorporate it in the future (at present the agent has simply one intention, in our example to keep the world clean from waste).

The class `CleanerAgent` inherits all these common structures and adds the application specific part of the net.

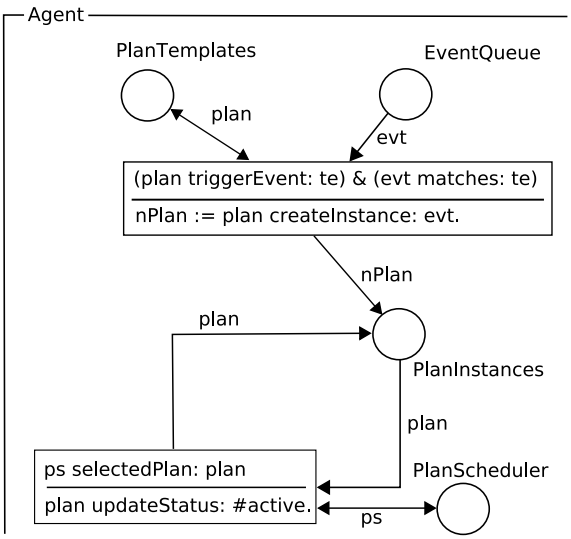

Figure 6. Part of the Agent's class object net.

That means that its object net serves also as the belief base (representation of the world described earlier). The agent's external actions are represented as methods of this class that can modify the belief base. Apart from these methods, predicates that test the belief base are also available for use by the plans.

5. Conclusions

In the paper, we have outlined the formalism of Object Oriented Petri Nets (OOPN), and we have discussed its usability in the field of deliberative agent modeling. This usability was demonstrated on the example of *Cleaner World* which has been implemented and tested in the PNtalk system. How we could see, OOPN allow us to model and simulate an agent and its behavior in proper way. During the work on the Cleaner world example we have also discovered some missing properties of the OOPN formalism which do not implicate problems on the expressive power of OOPN, but a description of some modeled aspects is more difficult.

In the future, we plan to create fully functional model of a complete BDI system using the OOPN formalism, and to extend OOPN for its missing properties making a modeling easier. Further research will also be focused on the *simulation based design* (SBD), thus the development of the system components (e.g. agents) based on modeling and simulation. The Cleaner world example is a first attempt to model agent systems using OOPN including a future possibility to apply principles of SBD. This activities follow on the research at our faculty and they will be provided under new projects which have started this year.

6. Acknowledgement

This work was supported by the Czech Grant Agency under the contracts GP102/07/P306, GP102/07/P431, GA102/05/H050 and Ministry of Education, Youth and Sports under the contract MSM 0021630528.

7. References

[1] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[2] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Agent Theories, Architectures, and Languages*, pages 155–176, 1997.

[3] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, USA, July 1987.

[4] V. Janoušek. PNtalk: Object Orientation in Petri nets. In *Proc. of European Simulation Multiconference ESM'95*. Prague, CZ, 1995.

[5] V. Janoušek and R. Kočí. PNtalk: Concurrent Language with MOP. In *Proceedings of the CS&P'2003 Workshop*. Warsaw University, Warsawa, PL, 2003.

[6] V. Janoušek and R. Kočí. Towards an Open Implementation of the PNtalk System. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation*. EUROSIM-FRANCOSIM-ARGESIM, Paris, FR, 2004.

[7] V. Janoušek and R. Kočí. Towards Model-Based Design with PNtalk. In *Proceedings of the International Workshop MOSMIC'2005*. Faculty of management science and Informatics of Zilina University, SK, 2005.

[8] A. Kubík. *Inteligentní agenty (Intelligent Agents)*. Computer Press, Brno, Czech Republic, 2004.

[9] D. Moldt and F. Wienberg. Multi-agent-systems based on coloured petri nets. In *ICATPN*, pages 82–101, 1997.

[10] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *EXP – in search of innovation*, 3(3):76–85, 2003.

[11] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In R. van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[12] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *KR*, pages 439–449, 1992.

[13] Y. Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.

[14] D. Weyns and T. Holvoet. A Coloured Petri Net for a Multi Agent Application. In *Proc. of the Second International Workshop on Modelling of Objects, Components, and Agents (MOCA'02), Aarhus, Denmark, August 26-27, 2002 / Daniel Moldt (Ed.)*, pages 121–140. Technical Report DAIMI PB-561, Aug. 2002.

[15] M. Češka, V. Janoušek, R. Kočí, B. Křena, and T. Vojnar. Pntalk: State of the art. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components, and Agents*, pages 301–307. Hamburg, DE, 2006.