Simulation Based Design of Control Systems Using DEVS and Petri Nets

Radek Kočí and Vladimír Janoušek

Faculty of Information Technology, Brno University of Technology,
Bozetechova 2, 616 00 Brno, Czech Republic
{koci,janousek}@fit.vutbr.cz

Abstract. Current model-based design methodologies use executable semi-formal models allowing for transformations including code generation. Nevertheless, the code should be finalized manually and further development or debugging by means of prime models is impossible. The paper introduces an approach to the system design called Simulation Based Design which uses fromalisms of DEVS (Discrete-Event Systems Specification) and Object Oriented Petri Nets (OOPN) allowing for clear modeling, a possibility to check correctness by means of simulation as well as by formal verification. The approach is based on techniques such as incremental development in the simulation, reality-in-the-loop simulation, and model-continuity. The model is understood as an executable program valid through all development stages including the deployment (the target system).

1 Introduction

Current trend in systems design methodologies aims at an efficiency and safety of development processes as well as at the quality of resulted systems. The classic methodologies define the development process as a sequence of analysis, design, implementation, testing, and deployment. These methodologies are simple for its realization, but is not sufficient for design of complex systems, because the system requirements change during the development process and the classic methodologies do not support it very much. As a response to this the incremental and iterative design methodologies were developed, e.g., Unified Process or Agile Methodologies. They allow for concurrent execution of design phases, the changes are done by small steps, lays stress on testing, etc. These methodologies are more successful for ordinary applications, but have their limitations. The main problem is that the implementation and testing are separated from the design phase. For instance, the models are usually used in the design phases, but when we wish to test dynamic properties of developed system or to get the target system, we have to implement an executable prototype or system according to these models.

Another kind of methodologies which were investigated in the last decade are commonly known as *Model-Driven Software Development* or *Model-Based Design* (MBD). The most popular methodology is *Object Management Group's Model*

R. Moreno-Díaz et al. (Eds.): EUROCAST 2009, LNCS 5717, pp. 849–856, 2009.

Driven Architecture (MDA) [7] based on Executable UML [11]. An important feature of these methods is that they use executable models. In these methodologies of new generation, the designer creates models and tests their correctness by simulation, so that there is no need to make a prototype. They also allow for model transformations including code generation. Nevertheless, the code has to be usually finalized manually and these changes are not incorporated back to the models. It entails a possibility of semantic mistakes or imprecision between models and transformed code. Moreover, the further development, debugging and investigating of target application by means of prime models is impossible, so the significance of model has declined.

The paper introduces an approach to the software system design called *Simulation Based Design*. We understand it as an approach which combines the concepts of *model-continuity*, *incremental development in the simulation*, and *reality-in-the-loop simulation*. The model continuity makes a tendency towards an elimination of generating the source code from models [3,5]. The system is developed incrementally, models are being improved and are simulated in each design step (*incremental development in the simulation*). It is possible to simulate external components to test the system functionality. The next step is to exchange simulated components for their real realization and to test developed system in the real conditions (*reality-in-the-loop simulation)*. Finally, the developed models, in particular the control of the system logic, is deployed into the target system. The next important idea is multi-paradigm modeling. In the design, it is useful to benefit from special properties of different formalisms, which can be combined. The paper is aimed to the DEVS and Petri Nets formalisms which are used in a complement way. While OOPN is the object based formalism, DEVS serves as a component based framework for embedding other formalisms.

The presented concepts are supported by the tool named PNtalk/SmallDEVS [5,6] based on both formalisms. Its architecture results from principles of meta-level and reflective architectures [3,4] allowing models to reciprocal cooperates with the tool environment, to define and use special statements which are not directly included in the used formalism, etc., which enables model continuity and reality-in-the-loop simulation techniques.

2 DEVS

DEVS stands for Discrete Event System Specification. DEVS specifies a system hierarchically. A model can be specified as an atomic or as a coupled model. Coupled models consist of interconnected atomic and coupled models. This results in a model hierarchy of loosely coupled models.

Atomic model. An atomic model represents a simple, indivisible entity. The atomic models is defined as a structure:

$$M = (X, Y, S, ta, \delta_{int}, \delta_{ext}, \lambda)$$

where

- X (resp. Y) is the set of input events (resp. the set of output events);
- S is the set of states;
- $ta : S \longrightarrow R^{+}_{0,\infty}$ is the time advance function, that returns the lifetime of a state;
- $\delta_{ext} : Q \times X \longrightarrow S$ is the external input transition function, where $Q = \{(s, t_e) | s \in S, t_e \in [0, ta(s)]\}$ is the total state set and t_e is the time elapsed since last transition;
- $\delta_{int} : S \longrightarrow S$ is the internal state transition function;
- $\lambda : S \longrightarrow Y^{\varepsilon}$ is the output function where $Y^{\varepsilon} = Y \cup \{\varepsilon\}$ and ε denotes the silent event.

The system is always in some state $s \in S$. If no external event occurs, the system is staying in state s for $ta(s)$ time. If the elapsed time t_e reaches $ta(s)$, then the value of $\lambda(s)$ is propagated to the output and the system changes to state $\delta_{int}(s)$. If an external event $x \in X$ occurs on the input in time $s \leq ta(s)$, then the system changes its state to $\delta_{ext}(s, t_e, x)$.

The definition of the atomic model can be simply modified in such a way that it allows to use input and output ports. This makes the composition of models significantly easier from practical point of view.

Coupled model. Coupled models describe the system as a network of coupled components. These components can be atomic or coupled models. This is in fact a recursive hierarchical definition. In this way, output events of a component can become input events of another component. Zeigler [13] showed, that the DEVS formalism is closed under coupling, in other words, for every coupled model a equivalent atomic model can be constructed.

3 Object Oriented Petri Nets

Several attempts to combine Petri nets and objects has been done in the nineteens, for instance Object Petri Nets [10], Cooperative Nets [12], Nets-in-nets formalism [1]. They are supported by specialized tools like, e.g., Renew [9]. One of the formalisms covering advantages of Petri nets and object orientation is a formalism of Object Oriented Petri Nets (OOPN) [2]. Petri nets allow to describe properties of the modeled system in a proper formal way and the object-orientation brings structuring and a possibility of net instantiation.

The formalism of OOPN consists of Petri nets organized in classes. Every class consists of an object net and a set of dynamically instantiable method nets. Object nets as well as method nets can be inherited. Inherited transitions and places of object nets (identified by their names) can be redefined and new places and/or transitions can be added in subclasses. Inherited methods can be redefined and new methods can be added in subclasses. A token in OOPN represents either a trivial object (e.g., a number or a string) or an instance of some Petri net. A *class* is specified by an object net, a set of method nets, a set of synchronous ports, a set of negative predicates, and a set of message selectors corresponding to its method nets and ports.

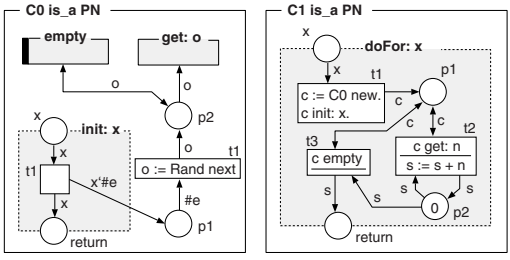

Fig. 1. An OOPN example

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

Method nets are similar to object nets but, in addition, each of them has a set of parameter places and a return place. Method nets can access places of the appropriate object nets in order to allow running methods to modify states of objects which they are running in. Method nets are *dynamically instantiated* by message passing specified by *transition actions*.

Synchronous ports are intended for synchronous interaction of objects. Synchronous port is a hybrid of method and transition that allow for synchronous interactions of objects. In order to be fired, the synchronous port has to be called (a method concept) and has to be firable (a transition concept). Synchronous port can be called only from a transition guard. The transition can be fired only if all called synchronous ports are able to fire. A special variant of synchronous port is *negative predicate*. Its semantics is inverted – the calling transition is firable if the negative predicate is not firable.

An example illustrating the important elements of the OOPN formalism is shown in Figure 1. There are depicted two classes *C0* and *C1*. The object net of the class *C0* consists of places *p1* and *p2* and one transition *t1*. The object net of the class *C1* is empty. The class *C0* has a method *init:*, a synchronous port *get:*, and a negative predicate *empty*. The class *C1* has the method *doFor:*. The semantics of the method *doFor:* execution is to randomly generate x numbers and return their sum.

4 Simulation Based Design

This chapter introduces basic principles of the simulation based design. For better understanding, the presented principles will be demonstrate on the simple robotic example. It is a part of the *leader-follower* example, which is one of classic robotic tasks consisting of at least two robots which are moving along the defined space. If two or more robots meet, they come to an agreement who will be the leader and the others follow him (reproduce his moving). If some robot

hits, e.g., the wall and is not able to continue in motion reproducing, he leaves the formation. The example uses Player/Stage software [8]. Player provides a network interface to a variety of robot and sensor hardware. Stage simulates a population of mobile robots moving in and sensing a two-dimensional bitmapped environment.

4.1 The Design Framework Based on OOPN and DEVS

As we have mentioned, the DEVS formalism serves as a component based framework for embedding other formalisms. Each DEVS component has the input and output ports which serve for communication between components. The component defined by the OOPN formalism delegates the communication responsibility to chosen OOPN object. This object (delegate) is just one in the component and its time-life is adherent to the component. The communication is provided via places of the object net—the port named `x` is mapped onto the place `x`. Let us show it on the example of the robotic system. The control mechanism is realized by OOPN and is wrapped onto a DEVS component named `PNAgent` (see the Figure 2 on the left). This component has three input ports which receive data from robot's sensors (`sonars`, `bumpers`, and `position`) and two output ports which reproduce generated commands (`rotateTo` and `move`). The delegate object is represented by the OOPN class `RCPlatform` (see the Figure 2 on the right). There are three special input places `sonars`, `bumpers`, and `position` corresponding with the component's ports. As soon as the data has been received to the input ports, they are placed to the matching places and the transition `createEvent` is fired. If the control mechanism makes a decision that, e.g., the robot has to rotate, the method `rotateTo:` is called, and the value representing a rotate degree is placed into the place `rotateTo`. This place is a special output place corresponding with the component's output port `rotateTo`. As soon as the value is placed to this port, it is accessible for another components which can read this port.

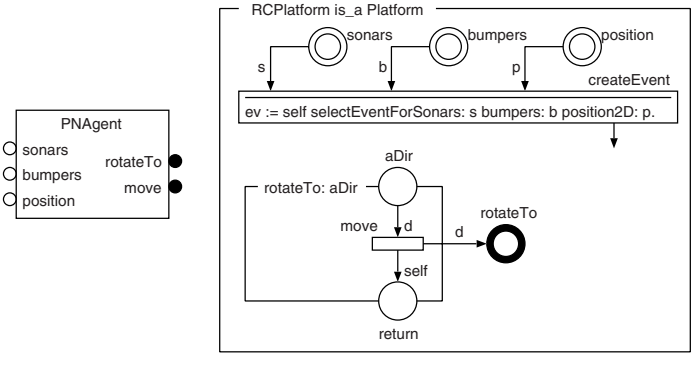

Fig. 2. DEVS ports and OOPN places mapping

4.2 System Design in Simulation

In the presented approach to the system development in simulation, we have to distinguish the system run and the simulation run. The simulation run allows to define and use special elements, e.g., we can use a special kind of places collecting statistics data. The implementation of that places is less efficient that the ordinary one, but it is no problem in the simulation run. We can also use simulated components instead of its real (software) variant. There are also three different kinds of time in which the model can execute—model time, real time, and quasi-real time. If the model is running in the model time, it runs in a pure simulation. If it is running in quasi-real time, the real time is a base for clock unit which is multiplied by defined constant. If it is running in real time, no time manipulation is supported—it is used for the deployed model (system).

Let us have a DEVS component of the robotic control. To test its correctness, we need either to implement the real communication between this component and the robot's sensors and actuators or to design their simulated variant. The second variant is much easier to realization so that the designer can quick and automatic test the component. The principle is shown in the Figure 3. The component `Test` simulates the robot (its state, position, etc.) and its neighborhood. It reacts to the `PNAgent` outputs and generates test data of sensors. The next component named `Stat` collects output data from other components and is able to trace the simulation run and to generate statistic data. Both `Test` and `Stat` components can be realized in the OOPN, DEVS, or other formalism.

The example of the `Stat` realization is shown in the Figure 4. It is implemented using the OOPN formalism and shows only a part of the class. Each input is stored in the place including the time stamp (see `self currentTime`)—the time can be real or model, it depends on the selected mode. Let us investigate the time which is consumed by `PNAgent` to make decision about the next step. If a new sensor data is generated (inject into the place `position`), the time stamp is stored to the place `p1`. The `PNAgent` indicates that the decision is made via the port `req`. Then the new time stamp is generated (the variable `t2`) and the consumed time is a difference between these two stamps (`t2-t1`).

To get statistic data we have to implement methods or make use the meta-level architecture of the PNtalk/SmallDEVS framework. It enables to define macros which are transformed into the relevant methods. The example is shown in the Figure 4. There is defined a macro `getMax` generating a method `getMax` which iterates for each object stored in the place `s2` (a pair `(t2,t1)` in our example)

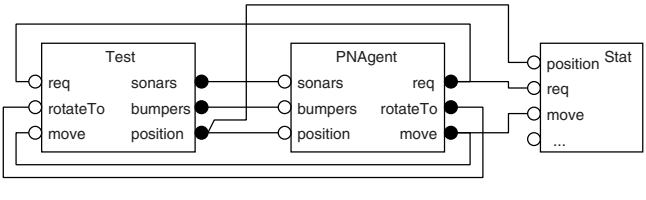

Fig. 3. Simulation Based Testing

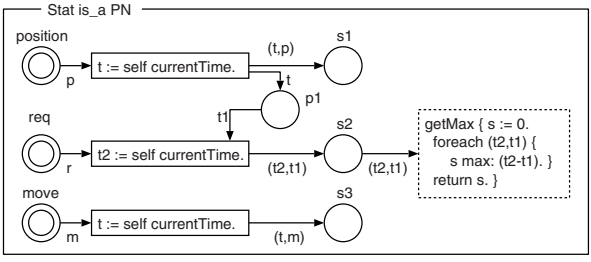

Fig. 4. Stat class in OOPN

and returns the maximum value of the consumed time (`t2-t1`). It very eases the design of such a kind of methods.

4.3 Model Continuity

The model continuity, together with reality-in-the-loop simulation, allows to use models in each development stages including the system deployment. Let us continue in our example. When we have tested the component `PNAgent`, we change a simulated communication (the component `Test`) for a real communication with the player/stage software (another DEVS component with the same interface). The components coupling will look the same like in the previous case (the Figure 3), just the component `Test` will have different implementation. Now we are able to test our system in real conditions using the same way (e.g., the component `Stat`). Finally, we remove all testing and simulated components, change the mode to the real time, and are able to deploy models to the system as is.

5 Conclusions

The paper has presented basic concepts of the *Simulation Based Design* technique. It uses formal models which can be simulated as well as deployed into the target system, makes a tendency towards an elimination of generating source codes from models, and uses various simulation techniques which ease the design and testing of particular components. A possibility to deploy models into the target system as is allows to debug system on the model basis—the system is always seen as a set of models which ease the debugging and further development of systems. The Simulation Based Design allows for high-quality and rapid development and results in more effective design process as well as in less numbers of mistakes and errors in the designed system. The further research will be aimed at the efficiency of the model execution in real time, the advancement of the tool supporting the presented design technique, and the usage of it in complex systems.

Acknowledgement. This work was supported by the Czech Grant Agency under the contracts GA102/07/0322, GP102/07/P306, and Ministry of Education, Youth and Sports under the contract MSM 0021630528.

References

1. Cabac, L., Duvigneau, M., Moldt, D., Rölke, H.: Modeling dynamic architectures using nets-within-nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 148–167. Springer, Heidelberg (2005)
2. Češka, M., Janoušek, V., Vojnar, T.: PNtalk - A Computerized Tool for Object Oriented Petri Nets Modelling. In: Proceedings of the 5th International Conference on Computer Aided Systems Theory and Technology – EUROCAST 1997, pp. 229–231. Las Palmas de Gran Canaria, ES (1997)
3. Janoušek, V., Kočí, R.: PNtalk: Concurrent Language with MOP. In: Proceedings of the CS&P 2003 Workshop. Warsaw University, Warsawa (2003)
4. Janoušek, V., Kočí, R.: 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)
5. Janoušek, V., Kočí, R.: 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)
6. Janoušek, V., Kočí, R.: Simulation and design of systems with object oriented petri nets. In: Proceedings of the 6th EUROSIM Congress on Modelling and Simulation, p. 9. ARGE Simulation News (2007)
7. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture – Practice and Promise, 1st edn. Addison-Wesley Professional, Reading (2003)
8. Kranz, M., Rusu, R.B., Maldonado, A., Beetz, M., Schmidt, A.: A player/stage system for context-aware intelligent environments. In: Proceedings of the System Support for Ubiquitous Computing Workshop, the 8th Annual Conference on Ubiquitous Computing (2006)
9. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for Petri nets: Renew. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 484–493. Springer, Heidelberg (2004)
10. Lakos, C.A.: From Coloured Petri Nets to Object Petri Nets. In: DeMichelis, G., Díaz, M. (eds.) ICATPN 1995. LNCS, vol. 935. Springer, Heidelberg (1995)
11. Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I.: Model Driven Architecture with Executable UML. Cambridge University Press, Cambridge (2004)
12. Sibertin-Blanc, C.: Cooperative Nets. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815. Springer, Heidelberg (1994)
13. Zeigler, B., Kim, T., Praehofer, H.: Theory of Modeling and Simulation. Academic Press, Inc., London (2000)