System Composition Using Petri Nets and DEVS Formalisms

Radek Kočí and Vladimír Janoušek

Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence
Brno, Czech Republic
{koci,janousek}@fit.vutbr.cz

***Abstract*—This paper is part of the work dealing with system developoment and deployment, where the system behavior should be modeled by formalisms allowing to define workflow scenarios and offering an interface for workflows synchronization. One such formalism is represented by Object Oriented Petri Nets (OOPN). OOPN are based on well-known class-based approach enriched by concurrency. Nevertheless, OOPN lacks one important element—a hierarchy followed by a simple way to model items exchanges on the fly. Therefore, the formalism of Discrete Event System Specification (DEVS) is taken into account. In the presented approach, the OOPN model is split up into DEVS components. Each component can be coupled with another component through the same compatible interface. A combination of OOPN and DEVS formalisms is used to compose the system using DEVS-based components, where each such component is modeled by means of OOPN. It preserves the advantages of using OOPN for behavior modeling and makes it possible to hierarchize models. The paper deals with the combination of both formalisms and compares the classic object approach to the component approach for system composition.**

***Keywords*-Object Oriented Petri Nets; DEVS; system composition; data passing.**

I. Introduction

The paper is part of the work dealing with system developoment and deployment [1][2][3], where the system is modeled, as well as implemented, by means of formal models. Present methods use various models in analysis and design phases, whereas these models usually serve as a system documentation rather than real models of the system under development. The system is then implemented in accordance with these models, whereas the code is either generated from models or implemented manually. Unfortunately, many implementation differ from the designed models because of changes created during the system debugging and improvement. Consequently, models become out of date and useless.

To solve this problem, the methodologies and approaches commonly known as Model-Driven Software Development were investigated and developed for many years [4][5]. These methods use executable models, e.g., Executable Unified Model Language (ExecUML) [6] in Model Driven Architecture methodology [7], which allow to test systems using models. Models are then transformed into code, but the resulted code has to often be finalized manually and the problem of imprecision between models and transformed code remains unchanged.

The system development methodology, which makes a base of the presented work, uses formal models for system description, as well as system implementation. Therefore, there is no need to transform models. Moreover, the system is developed using different kinds of models in simulation, i.e., it is possible to test systems in any state at any time. The combination of formalisms allows to derive benefits from their different features. This paper deals with two formalisms—Object Oriented Petri Nets (OOPN) [8], [9] and Discrete Event System Specification (DEVS) [10]. It preserves the advantages of using OOPN for behavior modeling and makes it possible to compose systems using DEVS-based components. This combination has been already used in previous works [2][11][12] as it is, without an analysis of its features and usefulness. This paper puts an accent on the ability of that concepts to model a system composition and its usability is demonstrated ising an example, which was defined in [11].

The paper is organized as follows. Section II deals with related work. Then, we briefly introduce the system in simulation concept in Section III and the used formalisms of OOPN and DEVS in Section IV. The different principles of system composition will be described in Section V, followed by analysis of system elements communication in Section VI. The usability of the presented approach is demonstrated in Section VII and the summarization and future work will be described in Section VIII.

II. Related Work

There are many works dealing with similar problems in the field of the design of control or embedded systems. The common feature is to use formal system (language, models, etc.) to software design and testing. There are two main motivations of formal system usage. First, to build and maintain control of the system in a quite fast and inuitive way. The High-level languages, especially based on Petri Nets, are used in this way. For example, the RoboGraph framework [13] for the robot application control uses hierarchical binary Petri nets for middleware implementation. In the area of

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

embedded systems, we can mentioned the work by Rust et al. [14], which uses Timed Petri Nets for the synthesis of control software by generating C-code, the work based on Sequential Function Charts [15], or the work based on the formalism of Nets-Within-Nets (NwN) [16][17][18].

These tools and works allow to *model* systems using a combination of different formalisms, but do not allow to use formal models in system *implementation*. The formalism of NwN is closest to the formalism of OOPN, but OOPN fully support an integration of formal description and programming language, which facilitates, e.g., reality-in-the-loop simulation or usage of formal models in the target application. The proposed approach allows to use of formal models to design, analyze and program applications, including a combination of simulated and real components. The main advantages are the following: there is no need for code generation, and the same formalisms and methods are used for further investigation of deployed systems.

III. System in Simulation Concept

The basic principles of the system development methodology [3][11] will be introduced in this section. The methodology supposes that the system is developed in the simulation; this concept will be outlined, too.

A. System Development Methodology

The modeling process is split up into three basic phases—identification of model elements, modeling the system behavior, and modeling the system architecture. The basic model elements are *subjects*, *roles*, and *activities*. The subject represents a data unit, e.g., the user working with the system or an individual element in the system. Each subject acts through its role. One subject can have more roles, e.g., the user can act as a reviewer, as well as a participant, in the conference review system. The activity represents the system functionality and is modeled by workflow scenarions. To model each such element, the formalism of OOPN is used. It can also be modeled by any other formalism allowing to define workflow scenarios and offering an interface for workflows synchronization, e.g., statecharts, activity diagrams, or other kind of Petri nets. The system architecture is modeled by classes that can be coupled into components using the formalism of DEVS.

B. Application Framework For System in Simulation

The used methodology [11] supposes that the system is being developed in the simulation. The development process starts with the empty simulation (simulation containing no model elements). Subsequently, in every subsequent step, model elements are being created, modified, or exchanged within the simulation. The simulation can be suspended, resumed, or restarted at any time, so that designers are able to test system behavior immediately, after each change. The model can be tested in real conditions, too. Therefore, a possibility to communicate with elements of product environment has to be ensured. The product environment is the target system where the developed model has to work.

The presented concept has to be supported by an application framework allowing to model the system, to simulate designed models, and to manipulate with models during the simulation. The application framework has to fulfil three basic requirements. First, to link models and product environment. Second, to work with models in simulations. Third, to exchange elements of models *on the fly*—the model elements should be exchanged with no changes in the depending model elements.

The application framework PNtalk, which satisfies previously listed requirements, has been developed [19]. Since the framework is implemented in Smalltalk [20], the objects of the OOPN formalism are directly available in the Smalltalk application and Smalltalk objects are directly available in the PNtalk framework. Nevertheless, OOPN objects can be linked to objects of any languages or formalisms allowing message passing.

Second, the PNtalk framework allows to execute models in different simulation modes that are suitable for design, testing, hardware/software-in-the-loop simulation, and system deployment. Using simulation allows, among others, to suspend (i.e., to exclude from the execution), to modify, or to exchange chosen parts of the model. By this point, we came in on the third requirement—a possibility to exchange model parts on the fly (any time during the system simulation). Therefore, the formalism of DEVS is taken into account. DEVS offers component approach allowing for wrapping an other kind of formalisms. The combination of OOPN and DEVS formalisms preserves the advantages of using OOPN for behavior modeling and makes it possible to hierarchize models. It allows the designer to derive benefits from component exchanges instead of object exchanges.

IV. Formal Models

We will briefly introduce the formalisms of OOPN and DEVS that make a base of the system development methodology [3][11].

A. Formalism of Object Oriented Petri Nets

The formalism of OOPN [21] is based on the well-known class-based approach. All objects are instances of classes, every computation is realized by message sending, and variables contain references to objects. This kind of object-orientation is enriched by concurrency. OOPN objects offer reentrant services to other objects and, at the same time, they can perform their own independent activities. The services provided by the objects, as well as the autonomous activities of the objects are described by means of high-level Petri nets—services by *method nets*, object activities by *object nets*.

Copyright (c) IARIA, 2014.

The formalism of OOPN contains important elements allowing to test object states (*predicates*) and to manipulate object state (*synchronous ports*) with no need to instantiate method nets. Object state testing can be negative (*negative predicates*) or positive (*synchronous ports*). *Synchronous ports* are special (virtual) transitions that cannot fire alone but need to be dynamically fused to some other transitions the synchronous port is called from (via message sending). *Negative predicates* are special variants of synchronous ports having inverted semantics—the calling transition is fireable if the negative predicate is not fireable.

For the sake of notation simplicity, we introduce the formal notation for following relationships. The term @ represents the relationship *is an instance of*. For example, $(a, C1) \in @$ means that a is the instance of the class named $C1$. We will write this relation in the form $a@C1$. If the instance identifier is not important, we will type only $@C1$. The terms $\overset{\mathcal{M}}{\rightharpoonup}$ and $\overset{\mathcal{M}}{\leftharpoondown}$ represent the relationship *contains a reference to*. For example, $a_1@A \overset{\mathcal{M}}{\rightharpoonup} a_2@B$ means that $a_1@A$ contains a reference to $a_2@B$ and $a_1@A \overset{\mathcal{M}}{\leftharpoondown} a_2@B$ means that $a_2@B$ contains a reference to $a_1@A$. If there are both relationships on the same elements, we will write $a_1@A \overset{\mathcal{M}}{\rightleftharpoons} a_2@B$.

B. Formalism of DEVS

The formalism of DEVS [10] can represent any system whose input/output behavior can be described as a sequence of events. The atomic DEVS model is specified as a structure M containing sets of states S, input and output event values X and Y, internal transition function δ_{int}, external transition function δ_{ext}, output function λ, and time advance function ta. These functions describe the behavior of the component.

This way, we can describe atomic models. Atomic models can be coupled together to form a coupled model CM. The later model can be employed itself as a component of a larger model. This way, the DEVS formalism brings a hierarchical component architecture. Sets S, X, Y are to be considered as structured sets. It allows to use multiple variables for specification of a state; we can use a concept of input and output ports for input and output events specification, as well as for coupling specification. In another words, components are connected by means of ports and event values are carried through these ports. We will denote input port by the notation $component_name \oplus port_name$ and output port by the notation $component_name \ominus port_name$.

As with the object approach, we will define the formal notation of DEVS components. Since the component represents the model description (cf. classes), as well as its executable form (cf. objects as instances of classes), there is no means for a notion *to be an instance*. The new component having the same structure and behavior of existing one can be created by *clonning* that component. To differ from the notation *contains a reference to* $\overset{\mathcal{M}}{\rightleftharpoons}$, we define the relationship *linked with* meaning that the component is linked with another one through their ports. This relationship will be represented by terms *left link* $\overset{\mathcal{D}}{\rightharpoonup}$, *right link* $\overset{\mathcal{D}}{\leftharpoondown}$, and *link* $\overset{\mathcal{D}}{\rightleftharpoons}$. For example, $c_1 \overset{\mathcal{D}}{\rightharpoonup} c_2$ (or $c_1 \overset{\mathcal{D}}{\leftharpoondown} c_2$) establishes a channel for data transmission from the component c_1 to the component c_2 (or from c_2 to c_1). The link $\overset{\mathcal{D}}{\rightleftharpoons}$ means there are both (left and right) links. To specify ports, we will write $c_1 \ominus port_1 \overset{\mathcal{D}}{\rightharpoonup} c_2 \oplus port_2$.

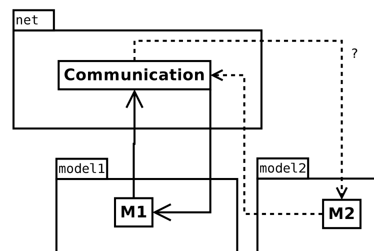

Figure 1. Packages with bidirectional relationship.

C. Combination of OOPN and DEVS Formalisms

The DEVS formalism offers a component approach, allowing to wrap other kinds of formalisms, so that each such formalism is interpreted by its simulator and simulators communicate with each other by means of the compatible interface. The OOPN model is then split up into components linked together by the compatible interface. Let $M_{PN} = (M, \Pi, map_{inp}, map_{out})$ be a DEVS component M, which wraps an OOPN model Π. The model Π defines an initial class c_0, which is instantiated immediately the component M_{PN} is created. Functions map_{inp} and map_{out} map ports and places of the object net of the initial class c_0. The mapped places then serve as input or output ports of the component, i.e., they are part of the component interface.

V. System Composition

The different principles of system composition will be described in this section. First, we describe the classic object oriented approach defining *packages*, where the interface is build up from classes or objects. Second, we describe the DEVS approach defining *components*, where the interface is build up from *ports*.

A. Packages Composition

In the classic approach, the interface of each package is built up from *classes* and their *operations*. Relationships between two packages should be only unidirectional—if there are bidirectional relationships, packages cannot be simply replaced by other packages. The example is shown in Figure 1. There are two packages `net` and `model1`; the class `model1.M1` needs to use the class

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

net.Communication and net.Communication notifies model1.M1 about incomming events—the relationship is bidirectional. If the package net would be used with another package (e.g., model2), there is a problem how to represent the association from net.Communication to model2.M2. The only way is *to change* the package net.

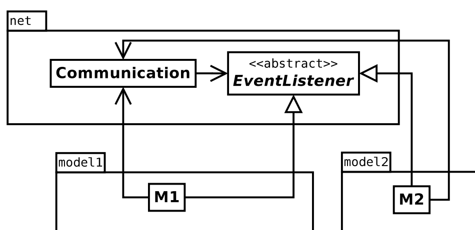

Figure 2. Packages with unidirectional relationship.

Figure 2 shows a solution of the previous situation—the class net.Communication depends on the newly created class net.EventListener and model1.M1 is derived from it—the relationship is unidirectional (only from model1 to net). If the package net would be used with another package, e.g., model2, this package only uses classes from net—no changes are needed. It is an application of known *Dependency Inversion Principle*.

B. Components Composition

In DEVS approach, the component interface is built up from *ports*. Relationships between two packages do not need to be only unidirectional; components can be simply replaced by other components in both cases. The example is shown in Figure 3. There are two components net and model1; the component model1 sends commands to the component net and net notifies model1 about incomming evets—the relationship is bidirectional.

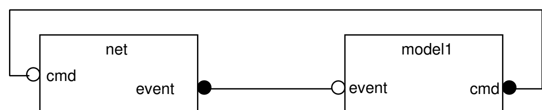

Figure 3. Components with bidirectional relationship.

If the package net would be used with another component (e.g., model2), there is no problem how to do it—the new component is simply re-connected through ports (see Figure 4).

VI. COMMUNICATON

The difference between objects and components replacement *on the fly* will be taken into account in this section.

A. Message Passing

In the classic approach, the package communication is provided by *message passing*. The object from one package (*client*) sends a message to the object from second package (*server*), whereas the client usually waits for an answer (until the message is processed—synchronous communication).

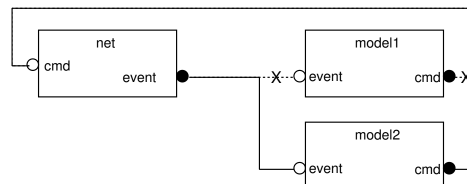

Figure 4. Components composition through ports with component changing.

Figure 5a) shows the communication between packages net and model1 through their interfaces. The instance $@net.Communication$ notifies the instance $@model1.M1$ about arrising events and $@model1.M1$ sends commands to $@net.Communication$. Let us assume that the class $Communication$ represents an interface to the real robot and the class $M1$ implements control algorithms. During the simulation, there can arise a need to test another algorithms in the current situation—the simple way is to change the control algorithm *on the fly* and continue in simulation. So, the component $model1$ is exchanged to $model2$; it follows that $@model1.M1$ is removed and $@model2.M2$ is put in its place.

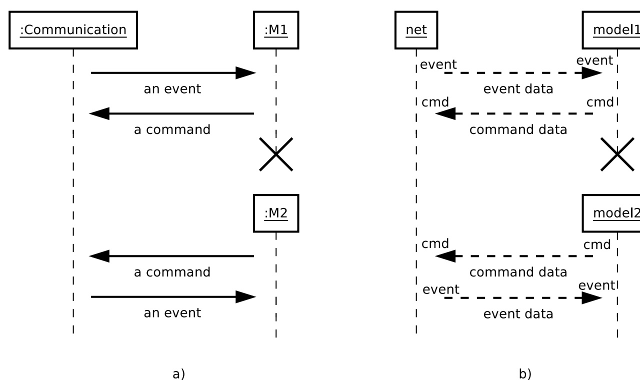

Figure 5. Communication through the object and component interfaces.

The object exchange *on the fly* means that one reference is exchanged to another one. To achieve this goal, the server has to be prepared for such an operation—it has to offer a protocol for attaching and detaching clients. Second, if the new client has a different protocol, it has to be adapted (cf., e.g., the design pattern *Adapter*). Third, if the detached component is in process, e.g., it processes a method which has been called from another object, the problem of its correct removing arrises.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

If we get back to the previous example, we make out that the instance of the class `net.Communication` has a reference to the instance of the class `model1.M1` and vice versa—from this point of view, there is a bidirectional relationship $@net.Communication \overset{\mathcal{M}}{\rightleftharpoons} @model1.M1$.

B. Data Passing

In DEVS approach, the component interface is built up from *ports*. Component communication is then provided by *data passing*; the *client* component sends a piece of data to the *server* component, whereas the client usually does not wait for an answer (asynchronous communication).

Figure 5b) shows a data passing between components `net` and `model1` through their interfaces. The component net notifies the component $model1$ about arrising events by carrying a piece of data from $net{\ominus}event$ to $net{\oplus}event$. The component $model1$ sends commands to the component net by carrying a piece of data through the data connection $model1{\ominus}cmd \overset{\mathcal{D}}{\rightarrow} net{\oplus}cmd$. Since the component represents the model description (cf. classes), as well as its executable form (cf. objects as instances of classes), there is no difference between components replacements during their composition or on the fly.

C. Comparison of Data and Message Passing

Since the Petri nets have good ability of describing processes, they can be used to model the difference between *message passing* and *data passing*. Figure 6a) describes the model of message passing and Figure 6b) describes the model of data passing.

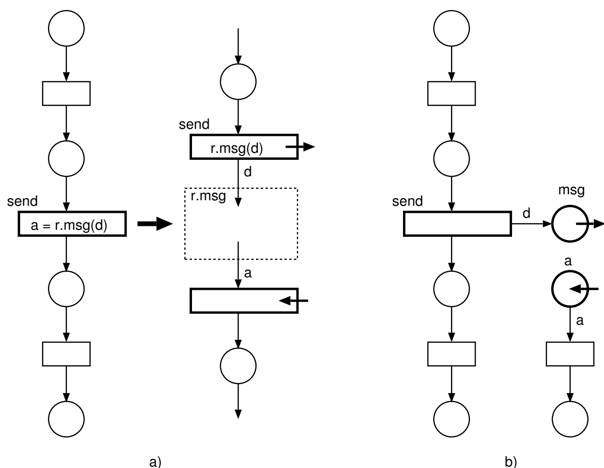

Figure 6. Comparison of message passing and data passing.

Let us investigate Figure 6a). The sequence of unnamed places and transitions represents one thread in the system behavior containing the message passing. It is modeled by the transition named $send$—it sends a message msg to the receiver r with a piece of data d and waits for the answer a. The transition $send$ can be split up into *output transition* (shown with output arrow), which needs to know receiver, message, and data, and *input transition* (shown with input arrow), which waits for the answer, i.e., waits until the called method $r.msg$ is finished.

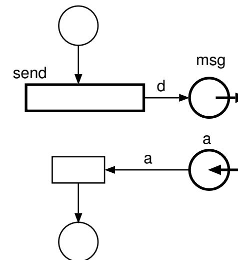

Figure 7. Adjusting data passing to sychronous communication.

Let us investigate Figure 6b). The transition $send$ needs to know only a piece of data d that are put to the *output port* msg (shown with output arrow) representing the message. First, the component does not care about the receiver (anything what is linked) and its interface. Second, the transition $send$ models asynchronous communication (no waiting for the answer). If the component needs to get the answer, it can define an independent thread, which is started by putting the answer to the *input place* a (shown with input arrow). In the case the component needs to wait for the answer, it can be modeled as shown in Figure 7.

VII. Data Passing and System in Simulation Concept

This section demonstrates the usability of the presented approach on a simple example of the robotic system, which has been described in [11]. The robotic system consists of the simulated robot (the data unit modeled by the subject *Device*), its role *Robot*, and one possible scenario of the robot behavior (the activity *Scenario*). These model elements are identified in accordance with development methodology (see Section III).

A. Model of Behavior

We will suppose a very simple activity, which can be described by the following algorithm: (1) the robot is walking; (2) if the robot comes upon to an obstacle, it stops, turns right and tries to walk, (3) if the robot cannot walk, it turns round and tries to walk; (4) if there is no possibility to walk, it stops. The activity net *Scenario* describing the presented behavior is shown in Figure 8.

The activity $@Scenario$ communicates with the object $@Role$, which is initially placed in the place *walking*. The communication is provided using *predicates* that serve for testing (see *isCloseToObstacle* and *isClearRoad*) and *synchronous ports* for action performing (see *stop*, *go*, and *turnRight*).

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

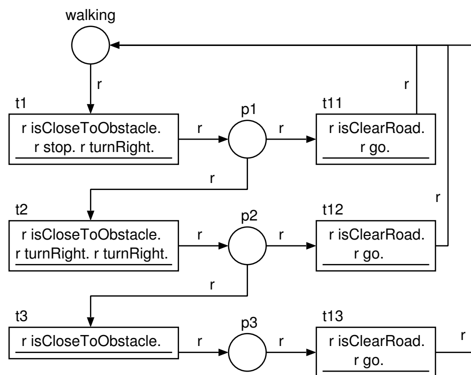

Figure 8. The activity net *Scenario*.

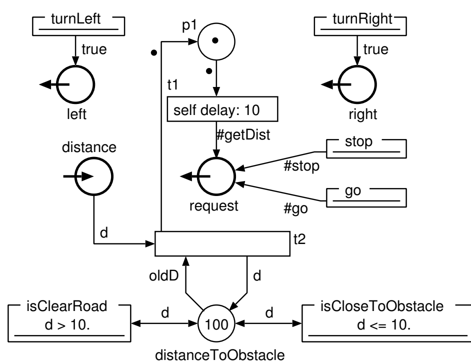

Figure 9. The role *Robot*.

The possible model of the role *Robot* is shown in Figure 9. The role offers information about robot's position by means of predicates *isClearRoad* and *isCloseToObstacle*. Moreover, the role offers synchronous ports *stop*, *go*, *turnRight*, and *turnLeft*, that represent commands forwarded to the subject.

B. Model of Composition

The system is composed of two components that are shown in Figure 10. The component *behavior1* consists of the role *Robot* and the activity *Scenario*—since these objects communicate using *message passing*, they have to be encapsulated into the same component. The component *robot* represents the *subject*, whose realization is schematically depicted in Figure 11.

The communication is provided using *data passing*. The role *Robot* represents the initial class c_0 of the component *behavior1*, so that the object $@Robot$ defines input and output ports. The component interface consists of output ports $\ominus request$, $\ominus left$, and $\ominus right$. We can see that appropriate synchronous ports only put a piece of data to these ports.

The component *robot* has input ports corresponding to the output ports of the component *behavior1*—their interfaces are compatible. Let us investige the following situation.

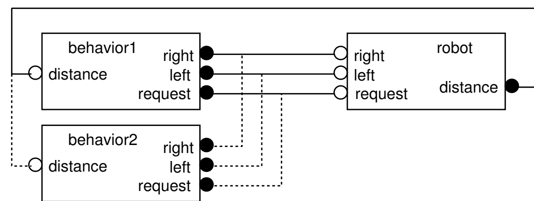

Figure 10. The robot system composition.

The activity $@Scenario$ requests turning right—it calls the synchronous port $@Robot.turnLeft$, which puts a value $true$ to the output port $behavior1\ominus left$. This value is carried through $behavior1\ominus left \overset{\mathcal{D}}{\rightarrow} robot\oplus left$ to the component *robot*, where performs advisable operations.

The role $@Robot$ checks actual robot's distance to the obstacle every 10 time units by requesting new data—it carries a symbol $\#getDist$ through $behavior1\ominus request \overset{\mathcal{D}}{\rightarrow} robot\oplus request$ to the component *robot*. The component *robot* gets a new information about the distance and carries it back through $robot\ominus distance \overset{\mathcal{D}}{\rightarrow} behavior1\oplus distance$.

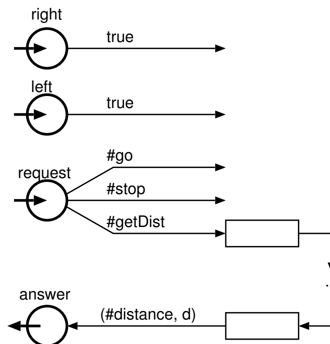

Figure 11. The subject component—an abstract view.

Anytime we need to exchange the model of behavior (for example, the actions change from *turning right* to *turning left*), we simply clone the existing component *behavior1*, the new component named *behavior2* will be created, we modify its realization and connect it through ports (see the component *behavior2* in Figure 10).

VIII. Conclusion and Future Work

This paper dealt with the usability of the OOPN and DEVS formalisms in the system development. The formalism of OOPN allows to define workflow scenarios and offers an interface for workflows synchronization. Nevertheless, it lacks a hierarchy followed by simple way to model items exchanges on the fly. Therefore, it was combined with the formalisms of DEVS, which offers hierarchized component approach—the OOPN model is split up into components linked together by the compatible interface. It preserves

the advantages of using OOPN for behavior modeling and makes it possible to hierarchize models.

This paper is part of the work dealing with system development and deployment using specific methodology and tool support. The application framework PNtalk, which satisfies required features, has been developed [19]. So far, it allows a communication to Smalltalk environment. Nevertheless, it can be linked to objects of any languages or formalisms allowing message passing. We plan to extend PNtalk to the Java and C/C++ platforms.

The proposed approach has one main disadvantage—the usage of application framework interpreting formal models, increases requirements on memory size and system performance. The future research will aim at efficient representation of choosed formal models and interoperability with another product environments. The application framework will be adapted to these conditions having lesser requirement for resources.

Acknowledgment

This work has been supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and by BUT FIT grant FIT-S-14-2486.

References

[1] R. Kočí and V. Janoušek, “System design with Object oriented Petri nets formalism,” in *The Third International Conference on Software Engineering Advances Proceedings ICSEA 2008*. IEEE Computer Society, 2008, pp. 421–426.

[2] R. Kočí and V. Janoušek, “OOPN and DEVS formalisms for system specification and analysis,” in *The Fifth International Conference on Software Engineering Advances*. IEEE Computer Society, 2010, pp. 305–310.

[3] R. Kočí and V. Janoušek, “Modeling and simulation-based design using Object-oriented Petri nets: a case study,” in *Proceeding of the International Workshop on Petri Nets and Software Engineering 2012*, vol. 851. CEUR, 2012, pp. 253–266.

[4] S. Beydeda, M. Book, and V. Gruhn, *Model-Driven Software Development*. Springer-Verlag, 2005.

[5] M. Broy, J. Gruenbauer, D. Harel, and T. Hoare, Eds., *Engineering Theories of Software Intensive Systems: Proceedings of the NATO Advanced Study Institute*. Kluwer Academic Publishers, 2005.

[6] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie, *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[7] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, ser. 17 (MS-17). John Wiley & Sons, 2003.

[8] M. Češka, V. Janoušek, and T. Vojnar, *PNtalk — a computerized tool for Object oriented Petri nets modelling*, ser. Lecture Notes in Computer Science. Springer Verlag, 1997, vol. 1333, pp. 591–610.

[9] R. Kočí and V. Janoušek, *Simulation Based Design of Control Systems Using DEVS and Petri Nets*, ser. Lecture Notes in Computer Science. Springer Verlag, 2009, vol. 5717, pp. 849–856.

[10] B. Zeigler, T. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. Academic Press, Inc., London, 2000.

[11] R. Kočí and V. Janoušek, “Object oriented Petri nets in software development and deployment,” in *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*. Xpert Publishing Services, 2013, pp. 485–490.

[12] R. Kočí and V. Janoušek, “Towards Design Method Based on Formalisms of Petri Nets, DEVS, and UML,” in *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, 2011, pp. 299–304.

[13] J. L. Fernandez, R. Sanz, E. Paz, and C. Alonso, “Using hierarchical binary Petri nets to build robust mobile robot applications: RoboGraph,” in *IEEE International Conference on Robotics and Automation*, 2008, pp. 1372–1377.

[14] C. Rust, F. Stappert, and R. Kunnemeyer, “From Timed Petri nets to interrupt-driven embedded control software,” in *International Conference on Computer, Communication and Control Technologies (CCCT 2003)*, 2003, pp. 124–129.

[15] O. Bayo-Puxan, J. Rafecas-Sabate, O. Gomis-Bellmunt, and J. Bergas-Jane, “A GRAFCET-compiler methodology for C-programmed microcontrollers, In Assembly Automation,” *Assembly Automation*, vol. 28, no. 1, pp. 55–60, 2008.

[16] R. Valk, “Petri nets as token objects: an introduction to Elementary object nets.” in *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, vol. 120. Springer-Verlag, 1998.

[17] D. Moldt, “OOA and Petri nets for system specification,” in *Object-Oriented Programming and Models of Concurrency*. Italy, 1995.

[18] L. Cabac, M. Duvigneau, D. Moldt, and H. Rölke, “Modeling dynamic architectures using nets-within-nets,” in *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA*, 2005, pp. 148–167.

[19] R. Kočí. PNtalk system. [Online]. Available: http://perchta.fit.vutbr.cz/pntalk2k [retrieved: August, 2014]

[20] A. GoldBerk and D. Robson, *Smalltalk 80: The Language*. Addison-Wesley, 1989.

[21] V. Janoušek and R. Kočí, “PNtalk: concurrent language with MOP,” in *Proceedings of the CS&P’2003 Workshop*. Warsaw University, Warsawa, PL, 2003, pp. 271–282.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4