The Object Oriented Petri Net Component Model

Radek Kočí and Vladimír Janoušek

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

***Abstract*—The formalism of Object Oriented Petri Nets (OOPN) is a part of the work dealing with the method of system development in simulation. The work is based on the idea that system models are always executed even if they contain only one simple element or any changes are performed. Moreover, this idea does not distinguish between system models, prototypes, or target system; everything should be presented by the same means. Nevertheless, it should be possible to use different formalisms to describe models. It follows that a common platform is needed. The platform has to be simple and has to allow to change models on the fly. The formalism of Discrete Event System Specification (DEVS) has been used to specify the platform, because it enables to compose system using components, whereas each such a component can be modeled by different formalism. Proposed approach preserves the advantages of using OOPN for behavior modeling of components and makes it possible to hierarchize models using DEVS-based platform. The paper defines a platform based on DEVS and OOPN formalisms and deals with a question of safe changes of components on the fly.**

Keywords–Object Oriented Petri Nets; DEVS; component platform; interface consistency.

I. INTRODUCTION

This paper is part of the *System in Simulation Development* (SiS) work [1] based on the formalism of Object oriented Petri nets (OOPN) [2]. The basic SiS principle consists in continuous incremental development of models in the *live system* with the goal to come to the target system without a need of implementation—there is no difference between models, prototypes, or target system. The SiS concept requires three basic conditions. First, models have to be able to combine different formalisms or languages, e.g., Petri nets and Smalltalk language. For instance, the control part of the developed system can be modeled by OOPN, which has to be able to communicate to sensors—the communication channel can be implemented in Smalltalk language. Second, models can be execute in different simulation modes that are suitable for design, testing, in-the-loop simulation, and system deployment [3]. Third, there has to be a possibility to exchange any elements of the models on the fly; the model elements should be exchanged with no changes in the depending model elements [4].

To achieve presented requirements, a *common platform* is needed. The platform has to be simple and has to fulfill the SiS requirements, mainly changing models on the fly. The formalism of Discrete Event System Specification (DEVS) has been used to specify the common platform. It enables to compose system using DEVS-based components, whereas each such a component is modeled by means of OOPN. It preserves the advantages of using OOPN for behavior modeling and makes it possible to hierarchize models.

So far, there have been works dealing with a usage of OOPN and DEVS formalisms, but the compact definition of common platform has not been introduced and a question about safe replacement has not been solved. The paper defines the OOPN component model based on the DEVS common platform to which the formalism of OOPN is incorporated. The question about component interfaces and their consistency during the component changes will also be discussed.

The paper is organized as follows. We describe concepts of the common platform in Section III. Then, we define the OOPN component model based on the common platform in Section IV. The Section V describes a problem of the component interface consistency and introduces interface constraints. Section VI deals with a realization of constraints based on the formalism of OOPN. The summary and future work is described in Section VII.

II. RELATED WORK

The modeling of software system in *live* environment is not new idea. Model-Driven Software Development [5][6] uses executable models, e.g., Executable UML [7], which allows to test systems using models. Models are then transformed into code, but the resulted code has to often be finalized manually and the problem with imprecision between models and transformed code remains unchanged. Further similar work based on ideas of model-driven development deals with gaps between different development stages and focuses on the usage of conceptual models during the simulation model development process—these techniques are called *model continuity* [8]. While it works with simulation models during design stages, the approach proposed in this paper focuses on *live models* that are used in target environments, i.e., when the system is deployed.

The research activities in the area of system changes on the fly are usually focused on direct or indirect approaches. The direct approach uses formalisms containing intrinsic features allowing to change the system. Formalisms are usually based on kinds of Petri nets. Reconfigurable Petri Nets [9] introduces a special place describing the reconfiguration behavior. Net Rewriting System [10] extends the basic model of Petri nets and offers a mechanism of dynamic changes description. This work has been improved [11] by a possibility to implement net blocks according to their interfaces. Intelligent Token Petri Nets [12] introduces tokens representing jobs by that the

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

dynamic changes can be easily modeled. Their disadvantage is that they usually do not define the modularity.

The indirect approach handles system changes using extra mechanisms. Model-based control design method, presented by Ohashi and Shin [13], uses state transition diagrams and general graph representations. Discrete-event controller based on finite automata has been presented by Liu and Darabi [14]. The presented methods use external mechanisms, nevertheless, most of them do not deal with validity of changes.

The approach presented in this paper combines direct and indirect methods. To define platform allowing to change component on the fly, the intrinsic features of the formalism of DEVS is used in combination with application framework allowing to work with simulation in live environment.

III. COMMON PLATFORM

As we mentioned above, we need to have a *common platform* allowing to interconnect different formalisms, as well as to change model element on the fly. We have decided [4] to use DEVS [15] approach to specify the platform. This section describes a formal base of the common platform and introduces a simple example to demonstrate its features and usage.

A. Discrete Event System Specification Platform

The formalism of DEVS can represent any system whose input/output behavior can be described as a sequence of events. The model consists of *atomic models* M. Their behavior is described by functions that work with input event values X and produce output event values Y. These functions are not important from the paper point of view, so that we will abstract them. Atomic models can be coupled together to form a *coupled model* CM. The later model can itself be employed as a component of a larger model. The atomic model, as well as the coupled model, corresponds to the term *component*. This way the DEVS formalism brings a hierarchical component architecture. The platform based on DEVS will be called *common component platform* and will be denoted $\mathcal{M}$. The set of components of the platform $\mathcal{M}$ will be denoted $\mathcal{D}$.

B. Component Interface

Sets X and Y of the component are usually specified as structured sets. It allows to define input and output ports for input and output events specification, as well as for coupling specification. Let us have the structured set $X = \{(v_1, v_2, \ldots, v_n) | v_1 \in X_1, \ldots, v_n \in X_n)\}$, where v_i represents a value of the ith variable from the domain set X_i. We will denote members $v_1, v_2, \ldots, v_n$ as *input ports* and will write $X = (V_X, X_1 \times X_2 \times \cdots \times X_n)$, where V_X is an ordered set of n *input ports*. The set of *output ports* V_Y is defined similarly on the structured set Y. The *component interface* is then built up from *input ports* V_X and *output ports* V_Y.

The component platform consists of components that are coupled through their ports. We define a relationship *coupling* $\overset{\mathcal{D}}{\rightharpoondown} \subseteq \bigcup_{i \in \mathcal{D}} V_Y^i \times \bigcup_{i \in \mathcal{D}} V_X^i$ meaning that there are channels for data transmission between ports of components. We will denote input port, resp. output port, by the notation $component_name \oplus port_name$, resp. $component_name \ominus port_name$. Then, the notation $c_1 \ominus p_1 \overset{\mathcal{D}}{\rightharpoondown} c_2 \oplus p_2$ means that there is the coupling between the output port p_1 of the component c_1 and the input port p_2 of the component c_2. The relationship $\overset{\mathcal{D}}{\rightharpoondown}$ can also be written in opposite direction $\overset{\mathcal{D}}{\leftharpoondown}$.

Then, the common component platform is defined $\mathcal{M} = (\mathcal{D}, \overset{\mathcal{D}}{\rightharpoondown}, V_X^{\mathcal{M}}, V_Y^{\mathcal{M}})$, where $V_X^{\mathcal{M}} = \bigcup_{i \in \mathcal{D}} V_X^i$ and $V_Y^{\mathcal{M}} = \bigcup_{i \in \mathcal{D}} V_Y^i$ represent ports that are accessible from the platform neighborhood.

C. Component Changes on the Fly

The component in the common platform is a model description, as well as its executable form. There is no difference between static and dynamic (live) representation of models. In comparison with classic object oriented approach, we need not care about classes, new instances, and reference changes (i.e., how to detach old objects and to attach new objects) at the moment of component changes. We simply create the new component and change the connections (couplings).

D. Example Specification

The concepts presented in the paper will be demonstrated on the small example consisting of sensor nets, a module collecting data from sensors, and a module making decision based on the data (the form of decision is not important). Other parts will be abstracted.

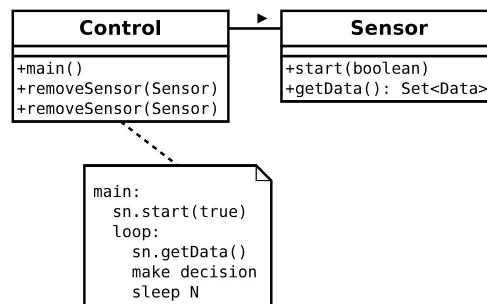

Figure 1. An example—class diagram.

In the classic object approach, we can define two analytical classes *Control* (decision maker) and *Sensor* (data collector). The class *Sensor* defines operations allowing to start or stop the data collection (*start*) and to get acquired data from sensors (*getData*). The class *Control* defines operations to attach and to detach a sensor (*addSensor* and *removeSensor*) and to control the process (*main*). The class diagram is shown in Figure 1. The basic algorithm of the method *main* is illustrated in the note window—it starts data collection and then performs following operations in the loop: gets data, makes decision, and waits for a while.

Now, we take the example specification to the common component platform. Figure 2 shows an example of the platform $\mathcal{M}_1$ containing two components $Control$ and $Sensors$, where $Sensors$ represents a communication channel to the sensor nets and $Control$ receives acquired data and makes decisions about the system. Due to simplification of notation, we will write cn and sn instead of full names.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

Then, the platform consists of $\mathcal{D}_1 = \{cn, sn\}$, where the components interfaces consists of $V_X^{cn} = \{data, run, stop\}$, $V_Y^{cn} = \{start, request\}$, $V_X^{sn} = \{start, request\}$, and $V_Y^{sn} = \{data\}$.

Now, we can compare DEVS-based common platform with object approach. First, the class $Sensor$ and the component sn. The method $start$ is represented by the input port $sn \oplus start$ receiving a command to start or stop data collecting. The method $getData$ is represented by a pair of input port $sn \oplus request$ and output port $sn \ominus data$. If any component asks for data, it puts a command to $sn \oplus request$ and the component reacts by putting data to $sn \ominus data$.

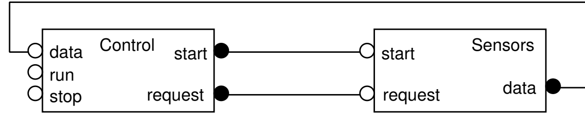

Figure 2. Common platform $\mathcal{M}_1$.

Second, the class $Control$ and the component cn. There is no port equivalent to the method $main$ because of intrinsic definition of the component behavior. Nevertheless, ports $cn \oplus run$ and $cn \oplus stop$ serve to start and to stop main loop of the component cn. These ports are not connected inside the platform; they will be used from outside to control platform run. The communication to cn surroundings is represented by ports $cn \ominus start$ (starting a data collection), $cn \ominus request$ (a request for acquired data) and $cn \oplus data$ (an answer for data requesting). The component cn sends commands to the component sn by carrying data through $cn \ominus start \overset{\mathcal{D}}{\rightharpoondown} sn \oplus start$ and $cn \ominus request \overset{\mathcal{D}}{\rightharpoondown} sn \oplus request$. The component sn reacts by sending data through the coupling $cn \oplus data \overset{\mathcal{D}}{\leftharpoondown} sn \ominus data$.

IV. OOPN COMPONENT MODEL

The common platform based on DEVS formalism offers component approach allowing to wrap another kind of formalisms, so that each such a formalism is evaluated by own means. The Object oriented Petri Net component model (OOPN component model) consists of DEVS components that are described by the OOPN formalism. This section introduces the OOPN formalism and its relationship to the *common platform* $\mathcal{M}$.

A. Object Oriented Petri Nets

First of all, let us agree upon the following definitions in the OOPN component system. The *Object oriented Petri net* is a tuple (Σ, c_0), where Σ is a system of classes and c_0 is an initial class. Σ contains sets of OOPN elements, which constitute classes. For the paper purpose, we will denote only selected elements that are used. The system of classes Σ is defined as follows $\Sigma = (C_{PN}, MSG, N_O, N_M, SP, NP, P, T)$, where C_{PN} is a set of OOPN classes, MSG is a set of message selectors, N_O is a set of object nets, N_M is a set of method nets, SP is a set of synchronous ports, NP is a set of negative predicates, P is a set of places, and T is a set of transitions. The message selectors MSG corresponds to method nets, synchronous ports, and negative predicates. Object nets describe possible autonomous activities of objects, while method nets describe reactions of objects to messages. A *class* C is defined as $C = (MSG^C, on^C, N_M^C, SP^C, NP^C)$, where $MSG^C \subseteq MSG$, $on^C \in N_O$, $N_M^C \subseteq N_M$, $SP^C \subseteq SP$, and $NP^C \subseteq NP$. Every net consists of places (a subset of P) and transitions (a subset of T).

The OOPN dynamics comprises the system of objects Γ. Elements from C describe a structure of simulation model and have to be instantiated to simulate the model. If the class $C \in C_{PN}$ is instantiated (the object o is created), the instance of object net on^C is created immediately. If the message $m \in MSG$ is sent to the object o, an instance of the method net is created. Then, we can define $\Gamma = (OBJ, INV)$, where OBJ is a set of objects including their object net instances and INV is a set of invoked method nets.

B. OOPN in Common Platform

In the common platform, there is no difference between a static representation of the model and its *live* (running, executed) form. To include the OOPN formalism, we introduce the *live model* of OOPN as the tuple $\Pi = (\Sigma, \Gamma, c_0, obj_0)$, where $c_0 \in C_{PN}$ is an initial class and $obj_0 \in OBJ$ is an initial object of the class c_0.

In the common platform, the OOPN model is split up into submodels, whereas each submodel has its own initial class c_0 and initial object obj_0. Let $M_{PN} = (M, \Pi, P_{c0}^{inp}, P_{c0}^{out})$ be a DEVS component M, which wraps an OOPN submodel Π. The initial class c_0 is instantiated immediately the component M_{PN} is created. The component interface (V_X, V_Y) is represented by subsets of places $P_{c0}^{inp}, P_{c0}^{out} \subseteq P$, where P is a set of object net places of the initial class c_0 and $P_{c0}^{inp} \cap P_{c0}^{out} = \emptyset$. There are bijections $map_{inp} : P_{c0}^{inp} \rightarrow V_X$ and $map_{out} : P_{c0}^{out} \rightarrow V_X$ mapping ports and places and the mapped places then serve as input or output ports of the component.

C. OOPN Example

Let us continue with the example from Figure 2. Figure 4 shows an OOPN model of the component *Sensors* (sn) and Figure 3 shows an OOPN model of the component *Control* (cn). Both models have the same basis—the loop driven by an external stimulus (a token placed in the place s).

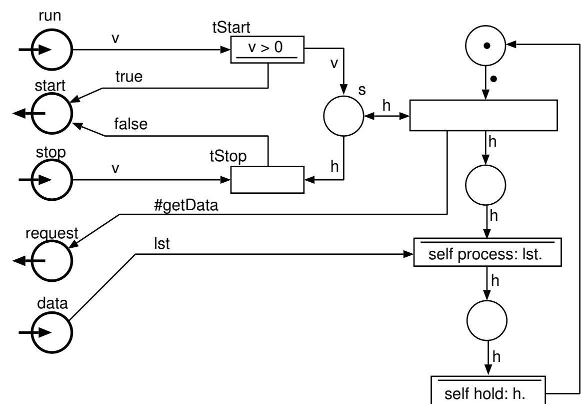

Figure 3. OOPN model of the component *Control*.

First, let us have a look at the component $Control$ (Figure 3). Input port $cn{\oplus}run$ expects a number h representing an interval of asking data from the component sn. Input port $cn{\oplus}stop$ expects any value—it only activates transition $tStop$, which suspends the loop. Both ports generate a command for coupled components through the output port $cn{\ominus}start$ (they put $true$ or $false$ to the mapped place $start$). The component cn asks for data by putting a symbol $\#getData$ to output port $cn{\ominus}request$ and waits for data (input port $cn{\oplus}data$). When the data are acquired, the method $process$: of the initial class c_0 is called and data are processed. Then, the loop waits for a given time unit h (the method $hold$:) and asks for data again.

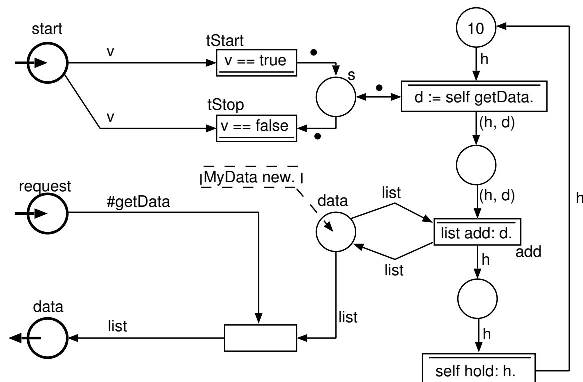

Figure 4. OOPN model of the component *Sensor*

Second, let us have a look at the component $Sensor$ (Figure 4). Input port $sn{\oplus}start$ expects values $true$ or $false$ activating transitions $tStart$ or $tStop$ that start or suspend the loop. The component receives a request for data by input port $sn{\oplus}request$ and puts data to the output port $sn{\ominus}data$.

The component sn acquires data in the loop, where the method $getData$ is called, the new data d is add by the transition add, and, finally, the loop waits for a given time unit h. The place $data$ contains an object (an instance of the class $MyData$) and the transition add simply adds new item by the method add:. Instance of the class $MyData$ is created and put into the place $data$ in the moment of object net instantiation (it is a place initialization, as shown in Figure 4).

D. Data Model Interface

So far, we did not care about actual data. Their form is not important for this paper (real numbers, integral numbers, etc.), only the way of data manipulation will be taken into account. It comes to this, that the data interface is important. Figure 5 shows identified interfaces and classes.

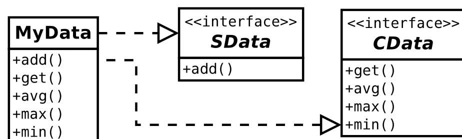

Figure 5. Classes and interfaces of data.

The component sn uses data storage by only way—adds a new item. So, the interface $SData$ containing the operation add: can be identified. Let us suppose, that the component cn needs following operations: get (getting an item), avg (average value), max (maximum value), and min (minimum value). These operations on data are performed within the method $process$:. Then, the interface $CData$ can be identified. The data storage (instance of the class $MyData$) used in the component sn (see Figure 4) has to implement the $SData$ interface. The storage object used in the component cn has to implement the $CData$ interface. Because both objects are identical (the object is carried through $sn{\ominus}data \overset{\mathcal{D}}{\rightarrow} cn{\oplus}data$), the class $MyData$ has to implement both interfaces, as shown in Figure 5.

V. Component Interface Consistency

The *System in Simulation* (SiS) concept assumes that components can be exchanged with no changes in the other components. In conjunction with the application framework [16], the component can be suspended, resumed, or changes any time during the system simulation. Therefore, it is necessary to be concerned with the problem of *component interface consistency*, in other words, the question whether component interfaces are compatible and whether its exchange is safe.

The component communication is provided by *data passing* [4]—the calling component (*client*) sends a data to the called component (*server*); the client does not need to wait for an answer. We will distinguish the structural aspect and the behavioral aspect of the component interface. The structural aspect is defined by ports and couplings. There is no problem to check if the components can be coupled or not. The behavioral aspect corresponds to the *concrete data* and their form.

A. Type constraints

Although the formalism of OOPN is pure object-based system and there is no need to define special kind of types instead of *class*, we will have special requirements to the set of types that can be checked:

- *a class* or *a subclass* – we need to check if the object is an instance of the class or its subclasses
- *an object interface* – we need to check if the object complies with the interface

Since we will check the type constraints, we have to define the term *type* in the context of the OOPN component system. CL_{env} is a set of classes from the product environment (the notation *product environment* is understood as the environment including language in which the application framework is implemented), $CL_{prim} \subset CL_{env}$ is a set of *primitive classes* (numbers, characters, and symbols), $I \subseteq \mathcal{P}(MSG)$ is a set of object interfaces, and ε represents a special kind of type meaning *unspecified type*. We define the interface I in the general way, as a set of operations that are independent from classes. The type is then $TYPE = CL_{PN} \cup CL_{env} \cup I \cup \{\varepsilon\}$.

Let T_P be a surjection $T_P : P \rightarrow \mathcal{P}(TYPE)$ assigning a set of types to a given place. The type of the place can be derived from the associations between classes, whereas there is no necessary to define only one type (and, thus, to allow all subtypes), but the set can be extended to next types. Implicitly,

each place has assigned the type ε. To discriminate between different levels of type constraints, we introduce following operators based on T_P:

- $\succeq$: $OBJ \times TYPE$ meaning the object is an instance of the class or derived classes, $\forall o \in OBJ : o \succeq \varepsilon$
- $\succ$: $OBJ \times I$, meaning the object complies with the interface, $\forall o \in OBJ : o \succ \varepsilon$

Let us continue with the example defined in Section IV-C. Figure 6–a shows type constraints defined on the input ports $stop$ and $data$ of the component $Control$ (cn). The port $stop$ requires any value of any type, so that the constraint is set to ε. The port $data$ requires objects of the class $MyData$, so that $\forall o$ in the place $data : o \succeq MyData$. The constraint will be written $\succeq \{MyData\}$.

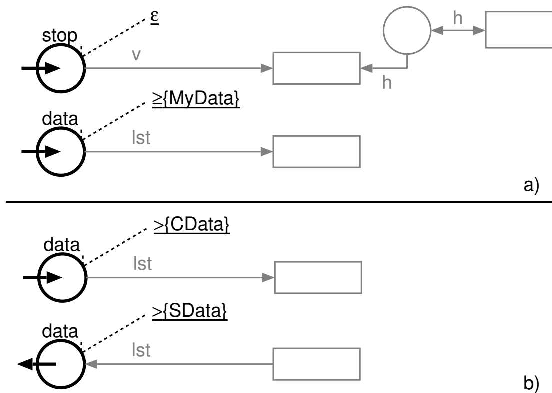

Figure 6. An example of type constraints.

Let us investigate a variant of interface usage—it is shown in Figure 6–b. There are depicted the input port $Control.data$ and the output port $Sensor.data$. Each of them operates with the different interface. $Control.data$ demands objects understanding methods defined by the interface $CData$, whereas $Sensor.data$ offers object understanding methods defined by the interface $SData$ (interfaces are discussed in Section IV-D).

B. Data constraints

Since the interface of the *common platform* is based on the principle of *data passing*, there will often be a request for constraints on data. First, let us define two auxiliary notions. Let $\mathcal{I}_G$ be a function $\mathcal{I}_G : TYPE \rightarrow \mathcal{P}(TYPE)$ assigning a set of generalized classes to the given class and $\mathcal{I}_S : TYPE \rightarrow \mathcal{P}(TYPE)$ be a function assigning a set of specialized classes to the given class. Then, $CL_{prim} = \mathcal{I}_S(Number) \cup \mathcal{I}_S(Character) \cup \mathcal{I}_S(Symbol)$. To discriminate between different levels of data constraints, we introduce following notions:

- *an enumeration* $\eta = \{e_1, e_2, \dots\}$, to check if the object o gets one of the listed values, $o \in \eta$; it can be used for symbols, numbers, or characters CL_{prim}
- *an interval* $\iota(i_1, i_2)$, to check if the object o gets a value from the interval, $o \in \iota(i_1, i_2)$; it can be used for numbers $\mathcal{I}(Number)$; there is a special value ω represents a maximal value or infinity

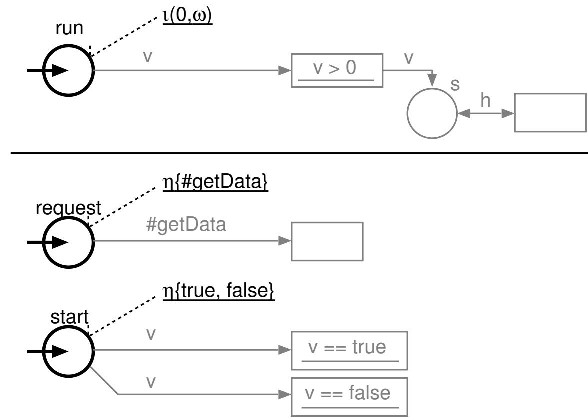

Figure 7. An example of data constraints.

Let us continue with the example defined in Section IV-C. Figure 7–a shows data constraints defined on the input port $cn \oplus run$. It requires a number from interval $\iota(0, \omega)$. Figure 7–b shows data constraints defined on the input port $sn \oplus (request, start)$. They require an enumeration of symbols or boolean values.

VI. Constraints Realization

Although the OOPN classes bring more intuitive modeling of behavior, they do not offer intrinsic definitions of *constraints* such as invariants or type checking. Nevertheless, there is very simple way how to define and test these conditions by means of OOPN [17]. Tests are generated by the application framework in accordance to required constraints defined on ports.

A. Type Constraints Testing

The test of *class constraints* is defined as $\theta_{\succeq}(p, ET) = \exists x \in p \wedge \nexists t \in ET : x \succeq t$, where x is an object in the place p and ET is a set of expected types. The test of *interface constraints* is defined as $\theta_{\succ}(p, ET) = \exists x \in p \wedge \nexists t \in ET : x \succ t$, where x is an object in the place p and ET is a set of expected types.

Both tests are implemented by negative predicates as shown in Figure 8. It follows the example defined in Section IV-C and shows two possibilities. First, the type constraint $\succeq \{MyData\}$ is defined for the input place $data$ of the component $Control$. This notion is equivalent to $\theta_{\succeq}(Control.data, \{MyData\})$. There is generated negative predicate $cTypeData$ and associated place ET containing a set of names of expected types. Names are stored in the form of symbols.

Firablity of the negative predicate is defined in two cases as follows. First, it is firable if there is no object in the associated place. Second, it is firable if the place is not empty and there is at least one object, which does not satisfy predicate conditions—on other words, the negative predicate finds all objects x that do not satisfy conditions. The condition is represented by arc expression t and calling special method $isKindOf$: t on the object x. The method is a part of object's *metaprotocol* and resolves in *true* or *false* depending

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

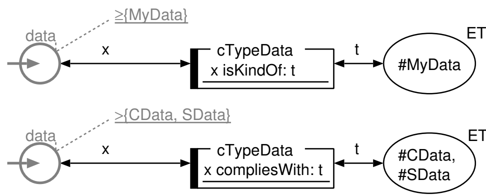

Figure 8. Type constraints realization.

on decision if the object is an instance of the class t (or its subclasses) or not. So, the predicate $cTypeData$ is *firable* if there is an object in the place $data$ and this object is not the instance of $MyData$.

Second possibility represents the type constraint $\succ \{CData, SData\}$ defined for the input place $data$ of the component $Control$. This notion is equivalent to $\theta_{\succ}(Control.data, \{CData, SData\})$. The constraint realization is the same as for $\succeq$ except that it uses the method $compliesWith$: instead of $isKindOf$:.

B. Data Constraints Testing

The tests of *data constraints* are implemented by negative predicates as shown in Figure 9. It follows the example defined in Section IV-C and shows two possibilities. First, the data constraint $\iota(0, \omega)$ is defined for the input place $start$ of the component $Control$. There is generated negative predicate $cDataStart$ having a condition corresponding to the defined interval. The predicate is firable if the condition is *not* satisfied.

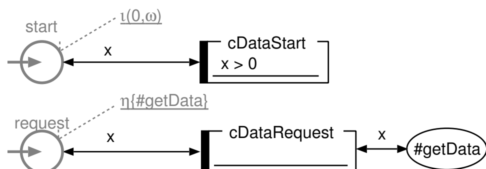

Figure 9. Data constraints realization.

Second, the data constraint $\eta\{\#getData\}$ is defined for the input place $request$ of the component $Sensor$. There is generated negative predicate $cDataRequest$ and associated place containing a set of expected symbols. The predicate is firable if there is found a symbol in the place $request$ that is not in the predefined set.

C. Exceptions

Constraints realizations presented in previous sections cannot be evaluated without calling them. Therefore, the new element of *exception* is introduced to the formalism of OOPN. The exception is demonstrated on the example of type constraint $\succeq \{MyData\}$ from Figure 8. The syntax is shown in Figure 10-a. The exception checks type constraint and if the constraint is not satisfied, it removes an object from the place $data$ and the associated *"any action"* is performed. The implementation is shown in Figure 10-b.

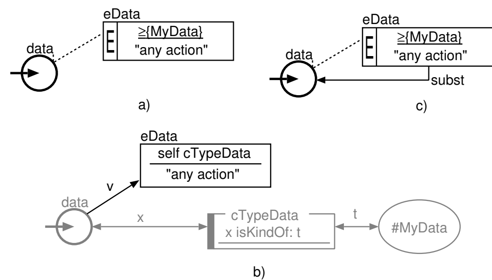

Figure 10. Invariants and testing conditions.

The exception may also have a side effect, e.g., it may offer substitute object and place it back to the place $data$ (shown in Figure 10-c).

D. Example of Component Changes

Let us continue with the example of common platform presented in Figure 2. The component $Sensors$ will be replaced by the component $Sensors2$ having the same structural interface, i.e., the same sets of input and output ports, as shown in Figure 11.

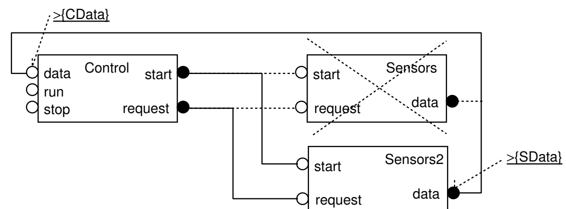

Figure 11. Common platform $\mathcal{M}_2$.

Let us suppose that the class $MyData$ has been changed to $MyData2$ containing only add: and get operations. Now, we will be only interested with $sn\ominus data \overset{\mathcal{D}}{\rightarrow} cn\oplus data$ coupling. The constraint $\succ \{SData\}$ is satisfied (operation add:), but the constraint $\succ \{CData\}$ is not satisfied (operations avg, max, and min are not present). Watching such incorrect changes, that do not have to be simply detected, allows to prevent systems from unexpected behavior.

VII. Conclusion and Future Work

The paper dealt with the concept of component platform based on DEVS and OOPN formalisms. It defined component interface and constraints above input and output ports. The interface is described by the means of OOPN places. Although they has assigned no type, for constraint testing it is possible to assign a set of types or constraints the objects have to satisfy. The concept of exception has been introduced to OOPN. Exceptions are a form of interface constraint testing. Incorrect changes done inside components do not have to easily be in evidence at the interface level. Constraints together with exceptions in languages that do not work with types allow to safe modification and changing component.

Future work will be aimed to a possibility to derive a set of types or constraints from the model analysis or simulation.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

The component interface will be also generalized to other formalisms that can be incorporated into common DEVS platform.

ACKNOWLEDGMENT

This work has been supported by the internal BUT project FIT-S-14-2486 and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070).

REFERENCES

[1] 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.

[2] 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.

[3] R. Kočí and V. Janoušek, "Formal Models in Software Development and Deployment: A Case Study," International Journal on Advances in Software, vol. 7, no. 1, 2014, pp. 266–276.

[4] R. Kočí and V. Janoušek, "System Composition Using Petri Nets and DEVS Formalisms," in The Ninth International Conference on Software Engineering Advances. Xpert Publishing Services, 2014, pp. 309–315.

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

[6] 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.

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

[8] D. Cetinkaya, A. V. Dai, and M. D. Seck, "Model continuity in discrete event simulation: A framework for model-driven development of simulation models," ACM Transactions on Modeling and Computer Simulation, vol. 25, no. 3, 2015.

[9] S. U. Guan and S. S. Lim, "Modeling adaptable multimedia and self-modifying protocol execution," Future Gener. Comput. Syst., vol. 20, no. 1, 2004, pp. 123–143.

[10] M. Llorens and J. Oliver, "Structural and dynamic changes in concurrent systems: Reconfigurable petri nets," IEEE Transactions on Automation Science and Engineering, vol. 53, no. 9, 2004, pp. 1147–1158.

[11] J. Li, X. Dai, and Z. Meng, "Automatic reconfiguration of petri net controllers for reconfigurable manufactoring systems with an improved net rewriting system based approach," IEEE Transactions on Automation Science and Engineering, vol. 6, no. 1, 2009, pp. 156–167.

[12] N. Q. Wu and M. C. Zhou, "Intelligent token petri nets for modelling and control of reconfigurable automated manufactoring systems with dynamic changes," Transactions of the Institute of Measurement and Control, vol. 33, no. 1, 2011, pp. 9–29.

[13] K. Ohashi and K. G. Shin, "Model-based control for reconfigurable manufacturing systems," in Proc. of IEEE International Conference on Robotics and Automation, 2011, pp. 553–558.

[14] J. Liu and H. Darabi, "Control reconfiguration of discrete event systems controllers with partial observation," IEEE Transactions on Systems, Man, and Cybernetics, Part B, Cybertetics, vol. 34, no. 6, 2004, pp. 2262–2272.

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

[16] R. Kočí and V. Janoušek, "The PNtalk System," 2015. [Online]. Available: http://perchta.fit.vutbr.cz/pntalk2k/

[17] R. Kočí and V. Janoušek, "Specification of UML Classes by Object Oriented Petri Nets," in ICSEA 2012, The Seventh International Conference on Software Engineering Advances. Xpert Publishing Services, 2012, pp. 361–366.