Creation of Adaptive User Interfaces Through Reconfiguration of User Interface Models Using an Algorithmic Rule Generation Approach

Benjamin Weyers
RWTH Aachen University
Virtual Reality Group
JARA-High Performance Computing
Seffenter Weg 23, 52074 Aachen, Germany
Email: weyers@vr.rwth-aachen.de

Nikolaj Borisov and Wolfram Luther
University of Duisburg-Essen
Department of Computer Science and
Applied Cognitive Science
Lotharstr. 63, 47057 Duisburg, Germany
Email: nikolaj.borisov@uni-due.de, luther@inf.uni-due.de

***Abstract*—Inefficient and error-prone interaction between human operators and technical systems was the reason for various catastrophic accidents in the past. User interfaces implement the communication between a human user and a technical system which is the reason why inaccurate design of user interfaces has been identified as one major factor for those errors. The use of adaptive user interfaces is one possible solution to reduce inefficient interaction by adapting the user interface to a specific user, task, or context. However, currently no self-contained formal approach exists that allows for the creation of adaptive user interfaces despite various advantages of formal methods: interaction becomes verifiable, formal methods close the gap between modeling and implementation by using executable formal languages, and they allow for using existing rewriting concepts making formal models adaptable. This paper introduces a new approach to a formal rule generation concept, which enables a flexible creation of adaptive user interfaces. This concept is based on a formal modeling and reconfiguration approach for the creation and adaptation of user interfaces. The applicability of this approach will be shown through an implementation of an adaptive user interface for adaptive automation. The main contribution of the presented work is a new concept for rule generation that is capable of adapting formally modeled user interfaces.**

Keywords–Formal Modeling; User Interface; Adaptive User Interface; Formal Reconfiguration; Rule Generation.

I. INTRODUCTION

Inefficient and error-prone interaction occurs during the use of interactive systems if the user interfaces are not sufficiently developed with respect to the needs and abilities of the user or user group [1]. As past events have shown, these errors can lead to catastrophic accidents, such as the disaster in Chernobyl [2]. Adaptive as well as adaptable user interfaces are primarily developed in order to increase human performance by changing the user interface according to a specific user, task, environment, context, or situation [3]. These kinds of user interfaces have shown a high potential in increasing usability [4], reducing errors in interaction [5], or in simplifying interaction with complex systems [6]. Furthermore, formal modeling approaches for user interfaces offer various advantages, like making it possible to directly execute or verify created models. Nevertheless, to our knowledge a self-contained and flexible formal approach for the implementation of adaptive user interfaces has not yet been discussed. Therefore, this work presents a formal modeling and reconfiguration concept for the creation of user interfaces that is extended by an algorithmic rule generation approach as a first step towards a generic and formal generation of adaptive (and adaptable) user interfaces.

The main difference between adaptive and adaptable user interfaces is the corresponding instance applying an adaptation to the user interface. An adaptable user interface is mainly changed manually by the user using tools. In contrast, adaptive user interfaces are changed by a technical implementation. For adaptive user interfaces, the type of adaptation is usually defined in an algorithmic fashion that is further parameterized by data selected from various sources, the interaction between user and the system, or from the system itself. Therefore the implementation of an adaptive user interface involves the type of user interface description or implementation on the one hand, and the adaptation concept that changes the user interface implementation and thereby influences the interaction, on the other. The description of a user interface can be divided into two parts: the *physical representation* and the *interaction logic* [7]. The physical representation covers all elements that are directly accessible by the user. In case of a classic graphical user interface, these elements can be buttons, sliders, or text fields. arranged in a specific layout. The interaction logic specifies the data-based communication between physical representation and the system to be controlled, as well as the logic and data-based dependencies between elements of the physical representation. Thus, interaction logic can also be denoted as the behavioral model of a user interface.

Based on this differentiation between the outward appearance of a user interface and its behavior, the adaptation (whether adaptable or adaptive) can be applied either on the physical representation, the interaction logic, or both. Using this concept combined with a set of tools for implementing an adaptable user interface, we conducted various evaluation studies. For instance in [5], we discussed a study that showed a significant error reduction in controlling a complex technical system by user-side adaptation of the physical representation and the interaction logic applied to an initial user interface model. The whole concept has been implemented based on

a formal modeling approach for user interfaces, which is based on this two-layered model. To this end, the physical representation has been formalized using an XML-based description while interaction logic has been formalized using a graph-based approach using reference nets, a variant of Petri nets. Furthermore, this has been combined with a formal rule-based reconfiguration concept, which is controlled through an interactive system by the user.

Apart from these aspects, formal modeling approaches offer further advantages to the modeling and reconfiguration of user interfaces. A formal user interface model can be directly executed using a simulator or interpreter. The combination of this approach with a rule-based reconfiguration technique creates a self-contained formal modeling approach. This supports the generation of adaptable and adaptive user interfaces, because a generic (formal) reconfiguration system can either be controlled by the user through an interactive system (as shortly described above) or by a system that generates rules algorithmically. This close integration of model-based creation and reconfiguration enables adaptation of user interfaces without losing the focus of creating computer-based systems [8]. This is a first step towards closing the gap between modeling and implementation of interactive systems [9]. Finally, this self contained approach of user interface modeling and reconfiguration prevents loss of information that can occur if a formal model is adapted by some informal or non-deterministic concept.

To extend this basic approach of formal adaptable user interfaces to the creation of adaptive user interfaces, the rule generation process necessary for the adaptation of user interface models has to be extended to enable computer-based and generic generation of reconfiguration rules. Therefore, the main concept introduced in this paper combines a description language for defining rule classes accompanied with a set of algorithms, which enable a software tool to instantiate a rule class based on a set of input data. A rule class further specifies a rule skeleton describing a basic structure of a rule that is to be instantiated. According to data defining and influencing the adaptation of a user interface, various channels can be identified, such as sensory data, a user model, or data generated by the controlled process (as discussed above). For instance, sensory data can be gathered to represent the context in which the interaction takes place. Nevertheless, various other types of data and data sources can be identified, which cannot be discussed completely in this paper. The main reason for this is that provided data is highly use case dependent. For instance, adaptivity of user interfaces can also introduce the user into the adaptation loop, such that she provides data or triggers the adaptation. Therefore, the presented work does not specify the type of data source but will support the description of various data types using a formal type specification language, called Resource Description Framework (RDF) [10]. This makes the definition (language) of rule classes independent from a specific use case by abstracting from the explicit data source to a data type that has to be delivered to the rule generation algorithm during runtime. Thus, the whole adaptation process becomes "semi-automatic" through the option of introducing the user into the loop.

Beside defining input data necessary for the instantiation of a rule class, various algorithms are discussed in this work offering functionality to the instantiation of rule classes. The aforementioned rule skeletons are defined based on grammars using nonterminal symbols for graphs and graph inscriptions. Thus, on the one hand, matching of nonterminal symbols in graphs and in inscriptions has to be performed based on the user interface model to be adapted, as well as on the input data as specified in the class and provided during runtime. On the other hand, algorithms for traversing a given graph are discussed regarding the extraction of certain parts in the user interface model that are part of its reconfiguration. Finally, changes of the visible part of the user interface have to be defined in the class and have to be finally applied to the user interface's physical representation. In conclusion, the main contribution of this work is an algorithmic approach for creating adaptive user interfaces based on a newly developed rule generation concept that defines the adaptation logic of such user interface models.

Before defining the semi-automatic rule generation, related work will be discussed in Section II identifying previous work done in the context of adaptive user interfaces, formal user interface modeling, reconfiguration, and adaptation. Section III describes a process to formal modeling and reconfiguration of user interfaces, which is executable and offers mechanisms for model-intrinsic adaptation through graph transformation systems. As computer-based adaptation of user interfaces assumes the accessibility of context information in various senses in a system's architecture as a formal model or description. Section IV presents a modeling approach of rule classes and an algorithmic rule generation concept that makes system-side adaptation of formal user interface models possible. In Section V, the whole adaptation process will be applied to the use case of adaptive automation and will show how the approach can be used in automated system control. Finally, Section VI will conclude the paper and will discuss future work aspects. The work at hand extends the previous paper by Weyers presented on the IARIA CENTRIC workshop 2013 [1]. Please consider that certain parts (Section V-B in particular) of this article have been reused in the work at hand to underline its origin.

II. Related Work and State of the Art

Adaptive user interfaces are nowadays an integral part of human-computer interaction research. Various works can be identified discussing different usecase dependent views to adaptive user interfaces, which have a similar goal: making interaction between a user and a system less error-prone and more efficient. Jameson [11] gives a broad overview of various functions of adaptive user interface that support this goal. One function he identifies is denoted as "supporting system use". He splits this category further up into the functions of "taking over parts of routine tasks", "adapting the interface", and "controlling a dialog", which are of main interest in the context of this work. Lavie et al. [12] identify certain dimensions of what data or knowledge is needed for the implementation of adaptive user interfaces: the task, the user, and the type of situation in which the interaction takes place. The latter, they characterize as routine vs. non-routine situations. Finally, they discuss the level of adaptivity that specifies the amount of adaptation applied to the user interface. These functions are provided by various implementations and work that has been done on adaptive user interfaces. A general overview of task and user modeling is given by Langley [13] and Fischer [14].

Nevertheless, this work does not focus on how the data and knowledge is gathered or described, but concentrates on how this data can be used for applying changes to a given user interface model.

However, various examples of the successful implementation of adaptive user interfaces can be found, which consider the former discussed aspects. For instance, Reinecke and Bernstein [4] describe an adaptive user interface implementation that takes cultural differences of users into consideration. They showed that users were 22% faster using this implementation. Furthermore, they made fewer errors and rated the adapted user interface as significantly easier to use. Cheng and Liu [15] discuss an adaptive user interface using eye-tracking data to retrieve user's preferences. Kahl et al. [16] present a system called SmartCart, which provides a technical solution for supporting a customer during her shopping process. It is able to provide context-dependent information and support, such as a personalized shopping list or a navigation service. Furthermore, in the context of ambient intelligent environments, Hervas and Bravo [17] present their approach of adaptive user interfaces, which is based on Semantic Web technologies. The so called ViMos framework is able to generate visualization services for context-dependent information.

All presented approaches and implementation have in common that they do not support a full-fledged formal modeling approach for the adaptation of user interfaces. Still, formal modeling approaches have certain advantages, as briefly discussed in the introduction. First, generated models can be directly executed. For instance, Navarre et al. [18] present their Interactive Cooperative Objects (ICO) approach, which is based on Petri nets. Using their interpreter called PetShop [19], generated models can be directly executed or simulated. ICO models mainly describe interaction logic in the sense discussed above. Barboni et al. [20] extended the ICO approach with a graphical user interface markup language, called UsiXML to define also the physical representation in a user interface model. UsiXML [21] is an XML-based user interface description language, that offers a "multi-path development" process, enabling the user to describe a user interface on different levels of abstraction. Still, UsiXML primarily defines the physical representation and only specifies, which sort of functionality is connected to it without describing it. Among others, the User Interface Markup Language (UIML) [22] is another XML-based markup language for describing user interfaces, which also excludes interaction logic from its description. Further formal modeling approaches can be found, such as the Petri net-based approach described by de Rosis et al. [23] or by Janssen et al. [24].

The second argument for the use of formal models is verification, using for instance, model checking or other formal verification methods. Brat et al. [25] discuss an approach using model checking to verify and validate formal descriptions of dialogs. This is of main interest, e.g., in modeling of user interfaces in safety critical situations [26]. Furthermore, Paterno and Santoro [27] discuss the use of formal verification in context of the investigation of multi-user interaction.

Finally, formal models of user interfaces can be used to apply reconfigurations to it and thus change their outward appearance, behavior, or both without necessarily leaving the formalization. Navarre et al. describe in [28] and [29] the reconfiguration of formal user interface models based on predefined replacements that are used in certain situations in safety-critical application scenarios, such as airplane cockpits. Blumendorf et al. [30] introduce an approach that changes a user interface during runtime. This approach is based on so-called "executable models", which combine design information and the current runtime state of the system. Interconnections between system and user interface are changed appropriately during runtime. Another approach that applies reconfiguration during runtime has been introduced by Criado et al. [31].

In conclusion, adaptive user interfaces play a central role in human-computer interaction and are still an ongoing research activity. Formal techniques in the development, creation, and reconfiguration are still discussed in research literature, offering various advantages regarding modeling, execution, and verification. Petri net-based as well as XML-based approaches are already applied in various application scenarios. Nevertheless, none of these approaches presents a full-fledged approach for the creation and reconfiguration of user interface models in one coherent formal modeling approach. Furthermore, none of the presented approaches discusses a closely related concept that enables computer-based systems to generate and apply reconfiguration in a flexible and usecase independent way. Therefore, this work introduces a self-contained approach for visual modeling and creation, rule-based reconfiguration, and algorithmic rule generation of user interfaces that builds a formal framework for the creation of adaptive user interfaces.

III. Formal Modeling of User Interfaces

As has been argued above, formal modeling of user interfaces offers various advantages, such as closing the gap between modeling and implementation. Nevertheless, formalization is often related to the use of complex description languages and requires a deep understanding of the whole formalization concept. The latter is addressed by solid documentation, which still needs a basic expert knowledge of a certain domain. In case of user interface modeling as introduced here, the modeler should have a basic understanding of programming languages and process modeling. Still, the problem of learning how to use a modeling language is mitigated by the use of a visual language paired with an intuitive point-and-click editor implementation.

The gap between modeling and execution is finally closed by a transformation of a given user interface model into reference nets, a special type of Petri nets. Using its associated simulator implementation (called Renew [32]), the whole model becomes executable, while rendering of the physical representation of the user interface is supported by a further software component, as implemented in the UIEditor (further discussed in Section III-E).

Another advantage of a formal representation of a user interface model is the possible close integration of reconfiguration concepts. This is achieved by using a rewriting concept applicable to reference nets. According to various reasons (as further discussed below in Section III-D), graph rewriting based on category theory has been chosen. Using this kind of rewriting, the rewritten user interface model does not have to be transformed in anyway reducing possible loss of information in the transformation. Finally, the rewriting

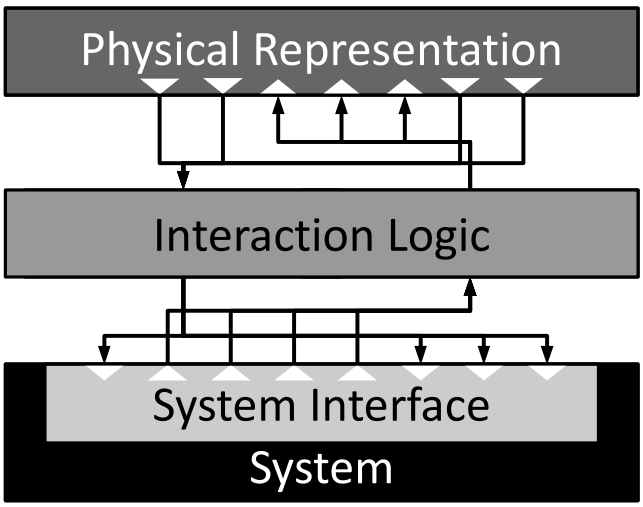

Figure 1. A two layered user interface model: the physical representation is directly accessed by the user, where the interaction logic specifies the data processing between physical representation and system to be controlled.

process, as well as the rewritten model is still verifiable in certain boundaries.

Therefore, this section introduces a formal modeling approach for user interfaces on a simple architectural basis, which is associated to a transformation algorithm that generates a reference net out of a created user interface model. After discussing a small example of the introduced modeling language and the transformation algorithm to reference nets, its associated reconfiguration concept will be introduced in more detail. Finally, this section will describe a tool called UIEditor that implements the modeling and reconfiguration concepts for user-driven, interactive creation and reconfiguration of user interface models.

A. Formal Interaction Logic Language - Formal Syntax

The basic concept of our approach for formal modeling of user interfaces relies on a two layered architecture that differentiates a user interface into its *physical representation* and its *interaction logic* (cp. Figure 1). In general, the term physical representation is not restricted to classic graphical user interfaces (GUIs) or WIMP (Windows, Icons, Menus, Pointers) interfaces [33]. Thus, a physical representation could also be a combination of speech recognition as input and a haptic device for output (this combination relates to multimodal user interfaces, such as described in [34]). Nevertheless, our work focuses on graphical user interfaces involving classic interaction elements, such as buttons, sliders, or text fields, as a first step implementation of the approach.

Interaction logic specifies the logical behavior of a user interface. It is defined by a set of processes that specify how data is processed that is emitted from the physical representation, such as events or inputted text, or from the system to be controlled. The system to be controlled can be specified as third layer but is not part of the user interface model (such as can be seen in Figure 1). Thus, interaction logic specifies the data processing, which takes place between the physical representation and the system to be controlled. These processes can be understood as graphs specifying data flow and data processing, also called *interaction processes*. Certain nodes in these graphs are dedicated to connect the process to the system or to the physical representation. Other nodes encapsulate complex data processing operations, such as casting of data types or arithmetic operations, and so forth.

We developed a visual and graph-based formal modeling language called *Formal Interaction Logic Language* (FILL) to support easy modeling capabilities for creating and editing interaction logic models in a visual editor (see Section III-E). Thus, FILL fulfills the requirement of providing visual modeling capabilities for creating interaction logic. First, the formal definition of FILL's syntax is given below (cp. [7]), which is partly based on nodes defined in the Business Process Model and Notation language (BPMN) [35].

Definition 1: the *Formal Interaction Logic Language* (FILL) is a 19-tuple

$$(S, I, C_I, C_O, P_I, P_O, X_I, X_O, B, T, P, E, l, g, c, t, \omega, \mathcal{L}, \mathcal{B}),$$

where S is a finite set of system operations, and I is a finite set of interaction-logic operations; P_I and P_O are finite sets of input and output ports; X_I and X_O are finite sets of input and output proxies; C_I is a finite set of input channel-operations; C_O is a finite set of output channel-operations; B is a subset of BPMN-Nodes, with

$$B = \{\oplus,\ \otimes,\ \odot\}. \tag{1}$$

S, I, C_I, C_O, P_I, P_O, X_I, X_O, T, and B are pairwise disjoint.

P is a finite set of pairs

$$\begin{aligned} P = \{(p,o) \mid p_I(p) = o\} \cup \{(p,o) \mid p_O(p) = o\} \cup \\ \{(p,o) | p'_I(p) = o\} \cup \{(p,o) | p'_O(p) = o\}, \end{aligned} \tag{2}$$

where $p_I : P_I \to S \cup I$ and $p_O : P_O \to S \cup I$ are functions with

$$\begin{aligned} \forall s \in S : (\exists_1 (p,s) \in P : p_I(p) = s) \wedge \\ (\exists_1 (p,s) \in P : p_O(p) = s),\ and \end{aligned} \tag{3}$$

$$\forall i \in I : \exists_1 (p,i) \in P : p_O(p) = i, \tag{4}$$

and where $p'_I : P_I \to C_I$ and $p'_O : P_O \to C_O$ are functions with

$$\begin{aligned} \forall c \in C_I : (\exists_1 (p,c) \in P' : p'_I(p) = c) \wedge \\ (\nexists (p,c) \in P' : p'_O(p) = c),\ and \end{aligned} \tag{5}$$

$$\begin{aligned} \forall c \in C_O : (\exists_1 (p,c) \in P' : p'_O(p) = c) \wedge \\ (\nexists (p,c) \in P' : p'_I(p) = c). \end{aligned} \tag{6}$$

E is a finite set of pairs, with

$$\begin{aligned} E = \{(p_O, p_I) \mid e(p_O) = p_I\} \cup \\ \{(p,b) \mid e'(p) = b,\ b \in B\} \cup \\ \{(b,p) \mid e'(b) = p,\ b \in B\}, \end{aligned} \tag{7}$$

where $e : P_O \cup X_O \to P_I \cup X_I \cup \{\omega\}$ is an injective function, ω is a terminator, and

$$\begin{aligned} \forall (p_O, p_I) \in E : (p_O \in X_O \Rightarrow p_I \notin X_I) \wedge \\ (p_I \in X_I \Rightarrow p_O \notin X_O), \end{aligned} \tag{8}$$

and where

$$e' : P_O \cup X_O \cup B \rightarrow P_I \cup X_I \cup B \cup \{\omega\} \tag{9}$$

is a function, extending e from basic FILL, and

$$\begin{aligned} \forall b \in B : (\#\{(p,b)|(p,b) \in E'\} > 1 \Rightarrow \exists_1 (b,p) \in E') \\ \vee (\#\{(b,p)|(b,p) \in E'\} > 1 \Rightarrow \exists_1 (p,b) \in E'). \end{aligned} \tag{10}$$

l is a function with

$$l : E' \rightarrow \mathcal{L}, \tag{11}$$

where $\mathcal{L}$ is a set of labels.
g is a function with

$$g : B \rightarrow \mathcal{B}, \tag{12}$$

where $\mathcal{B}$ is a set of Boolean expressions, also called guard conditions or guard expressions.
c is a relation with

$$c : C_I \rightarrow C_O. \tag{13}$$

T is a finite set of data types and t is a total function with

$$t : (P_I \cup P_O \cup X_I \cup X_O) \rightarrow T. \tag{14}$$

The visual representation of FILL's elements (syntax) is shown in Figure 2. FILL is mainly divided into four kinds of nodes (operation nodes, proxy nodes, BPMN nodes, and a terminator node) and two types of edges (data flow edge and channel reference edge), which are named according to the previous given syntax definition. *Operation nodes* are nodes that specify connections to the system (*system operation*), represent data processing operations (*interaction-logic operation*), or define relations between different subgraphs of the interaction logic (*channel operation*). Operation nodes are in general equipped with input and/or output ports. These connection points for edges represent data input or output. For instance, the shown example of an interaction-logic operation in Figure 2 consists of three input ports and one output port. Thus, the semantic of this operation is that it is executed if three data objects are sent to the three input ports via incoming edges. After the data processing function or method associated to the operation has been executed successfully, the result is passed back into the process via the single output port. In case of system operations, the inputted data object is passed to the system and/or a data object is returned from the system into the interaction process. Channel operations are further connected to each other by channel reference edges. Data objects sent to an input channel operation are redirected to one or more output channel operations as defined by channel reference edges. This enables FILL models to be modularized.

Another group of nodes are *proxy nodes* (Figure 2, upper right corner). These represent interaction elements that are part of the physical representation. Thus, proxy nodes are capable of sending data objects to the interaction process emitted by an interaction element or returning a data object from an interaction process to the associated interaction element. *BPMN nodes* as third group (Figure 2, right) define fusion and branching of interaction processes. Every node follows another type of fusion and branching semantics. An *AND* node branches an incoming interaction process by sending a copy of the incoming data object to every outgoing interaction process. In case of fusing different interaction processes, the outgoing process will be only triggered if all incoming interaction processes provide a minimum of one data object. An additional guard condition has to define which incoming data object will be copied to the outgoing interaction process, as can be seen in Figure 2. An *XOR* node has a contrary semantic to an AND node. In the branching case, exactly one outgoing process will be triggered by an incoming data object, which has to be further specified by a guard condition. In the fusion case, the XOR node simply redirects an incoming data object (whatever data process sent this data object) to the outgoing process. The *OR* node is a mixture of both AND and OR nodes. By specifying groups of incoming (fusion case) or outgoing (branching case) edges, an OR node behaves as an AND node concerning groups (every edge of a group has to provide a minimum of one data object in the fusion case, or every process of a group is triggered in the branching case, respectively), where edge groups are handled similarly to an XOR node among each other.

As far as formal languages are defined by formal syntax and semantics, formal semantics can be defined in two ways: (a) define the semantics of a formal language using mathematical formalism or (b) formally map a language's syntactic elements to another formal language that provides formal semantics. In case of FILL, a formal transformation to reference nets has been defined and algorithmically implemented. A reference net is a special type of Petri net, which has been equipped with formal syntax and semantics definitions [36]. It is a colored Petri net formalism that specifies an inscription language offering the definition of typed tokens and the specification of references to net instances. This mechanism makes it possible to instantiate nets and to assign resulting net instances to tokens using references. Furthermore, transitions can be inscribed with synchronous channels, which can be also used to call Java methods using the associated simulator Renew [32], which is implemented in Java. Java is further used to transform FILL to reference nets, where interaction-logic and system operations are implemented based on pre-defined interfaces. This enables Renew to call these functions through the interfaces' implementations. In general, a synchronous channel associates transitions with each other in such a way that they are only able to fire synchronously, which is also true for associated Java methods as discussed above.

B. Formal Interaction Logic Language - Formal Semantics through transformation to reference nets

The transformation of FILL to reference nets is algorithmically defined. The whole formal specification of the algorithm can be found in [7]. In this paper, the transformation will be discussed visually because the formal definition of the algorithm would exceed the scope of the paper.

Before starting the description of the transformations, some definitions are necessary to understand the inscriptions generated by the algorithm. To stay consistent to the original specification of the transformation algorithms, the definitions below have been extracted from the original sources [37] and [7] respectively.

Definition 2: Assume a given FILL graph as 19-tuple as defined above. Based on this, the functions f, κ, id, id_s can be defined as follows.

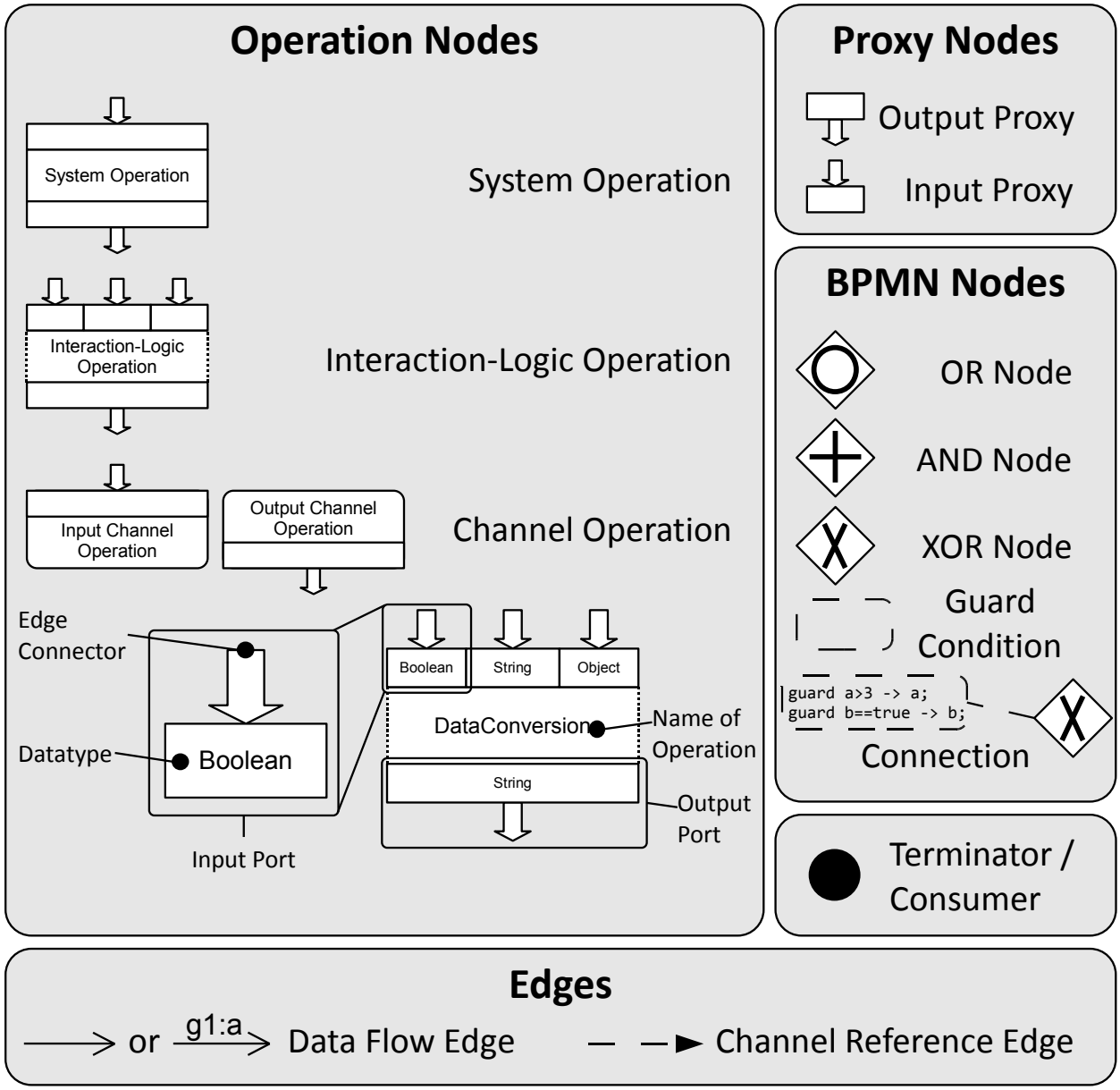

Figure 2. Visual specification of syntactical elements of the Formal Interaction Logic Language (FILL).

f is a function, with

$$f : S \cup I \cup C_I \cup C_O \rightarrow \mathcal{F}, \quad (15)$$

where $\mathcal{F}$ is a set of function calls. These function calls reference different types of underlying structures in the system or in the implementation of interaction-logic operations. Depending on the underlying programming language or system, these references have different syntaxes. Here, reference nets use Java method calls for calling code from the net.

κ is a function, with

$$\kappa : X_I \cup X_O \rightarrow \mathcal{I}, \quad (16)$$

where $\mathcal{I}$ is a set of references on interaction elements on the physical layer of the user interface.

id is a total bijective function, with

$$id : S \cup I \cup C_I \cup C_O \cup P_I \cup P_O \cup X_I \cup X_O \cup B \rightarrow \mathcal{ID}, \quad (17)$$

where $\mathcal{ID}$ is a set of ids, that identifies any node, port, or proxy in FILL. Based on the formal, graph-based definition of FILL, global identifiers are not necessary. In the transformation to reference nets and for representation in data formats like XML, ids play an important role.

$id_s : S' \rightarrow \mathcal{ID}$ is a total bijective function that matches a place in a reference net to an id similar to function id. $S' \subseteq S$ is a subset of places representing connections to and from a BPMN node. This function is necessary for the transformation of BPMN nodes; it compensates for the fact that a BPMN node does not have ports associated with ids. The inverse function id_s^{-1} matches an id to a place in a reference net. Due to the

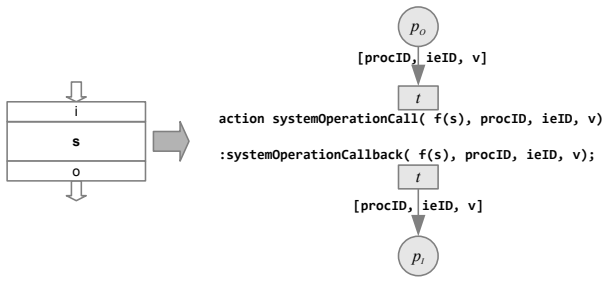

Figure 3. Transformation of a system operation into a reference net.

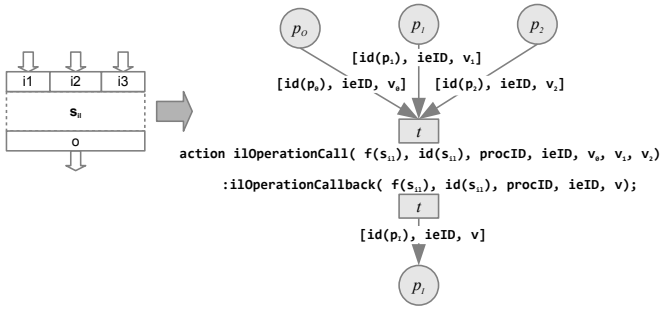

Figure 4. Transformation of a interaction-logic operation into a reference net.

bijectivity of id_s, there is an inverse function id_s^{-1}.

$ieID$ and $procID$ are ids that are generated in the transformation process. $ieID$ indicates the id associated to the interaction element that triggers or is triggered by the interaction process. $procID$ is used to further specify the data flow, as can be seen in the example discussed in Section III-C.

1) Transformation of Operation nodes: The transformation of interaction-logic or system operation nodes basically results in the generation of two transitions; one calling an associated (Java) method and one for reentering the net after the method returns. The inscriptions of these transitions only differ in the naming of the synchronous channel, which calls the associated method (systemOperationCall vs. ilOperationCall) and specifies the reentering point (systemOperationCallback vs. ilOperationCallback). They further differ in the number of variables that are send to the method (see Figure 3 and 4, respectively), which is indicated by its name $f(op)$, where op indicates the transformed operation node. Data values sent to and from operation nodes are associated to variables, here indicated with v and v_0 to v_2 in Figure 3 and 4.

The main difference between the transformation of a system operation node (as shown in Figure 3) and the transformation of an interaction-logic operation node (as shown in Figure 4) is the transformation of input ports. According to FILL's syntax definition, every interaction-logic operation has 0 or 1 output port and 0 to n input ports. In case of system operation nodes, there is exactly 1 input and 1 output port. In general, input and output ports are transformed into an edge/place combination as can be seen in Figures 3 and 4.

For the transformation of channel operations, channel edges have to be considered beside the operation nodes themselves. First of all, output channel operations are transformed into a transition-edge-place subnet as can be seen in the lower left corner of Figure 5. The transformation of an input channel operation is more complex. For every outgoing channel edge (connection), a place-edge-transition subnet (indicated as places q_0 to q_2 and transitions t_0 to t_2 in Figure 5) is generated, which is further connected to a main transition representing the operation (indicated as t_I in Figure 5). The connection between input and output channel operations is transformed into inscriptions, such as shown in Figure 5 for c_I and c_0, which are indicated by the used id of c_0 retrieved by $id(c_0)$. Both transitions are connected via a synchronous channel named $channel$. The keyword $this$ references to the current net instance, thus does not reference another net instance or external sources. If the shown example net is simulated, transition t is fired synchronously with transition t_0 according to the synchronous binding semantics of synchronous channels in reference nets.

2) Transformation of BPMN nodes: For BPMN nodes, the transformation into reference nets focuses even more on the structure of the generated net than it is the case for operation nodes. Here, the firing semantics of reference nets is actively used for modeling of the semantics of BPMN nodes as used in FILL. The semantics of BPMN nodes in FILL has been discussed above in Section III-A. Below, the transformations will be described per BPMN node in case of fusion and branching of interaction processes. For any transformation it is true that for any incoming and outgoing process a place is generated defining the entrance or exit point of the BPMN node, as can be seen in Figure 7.

AND(fusion): The outgoing process is only triggered if all incoming processes provide one or more data objects. This semantic is reflected in the structure of the reference net by defining the places representing the incoming processes as precondition for the transition t, which represent the BPMN node itself. The associated guard condition specifies which data object (here the object associated to the variable a) is copied to the outgoing process. In this case, the guard condition is obligatory in the FILL graph. The syntax of guard conditions has been specified compatible to guard conditions in reference nets, as specified in [36].

AND(branch): All outgoing processes will be triggered if the incoming process provides a data object. This semantic has been simply realized by specifying all places representing an outgoing process as postcondition of the transition representing the AND node. The shown guard condition in Figure 7 is optional and specifies under which condition the incoming data object is redirected to the outgoing processes.

XOR(fusion): Every incoming data object will be redirected to the outgoing edge by copying the data object to it. Therefore, for every incoming process one transition is generated that is optionally inscribed by a guard condition corresponding to the guard condition specified in the genuine FILL graph.

XOR(branch): Only one of the outgoing processes is triggered in case of an incoming data object. Therefore, any outgoing process is represented as a transition in the transformed reference net. Which outgoing process will be triggered has to be defined by an obligatory guard condition in the same sense as discussed above.

OR(fusion): Groups of processes can be defined by edge inscription in the FILL graph as can be seen in Figure 7.

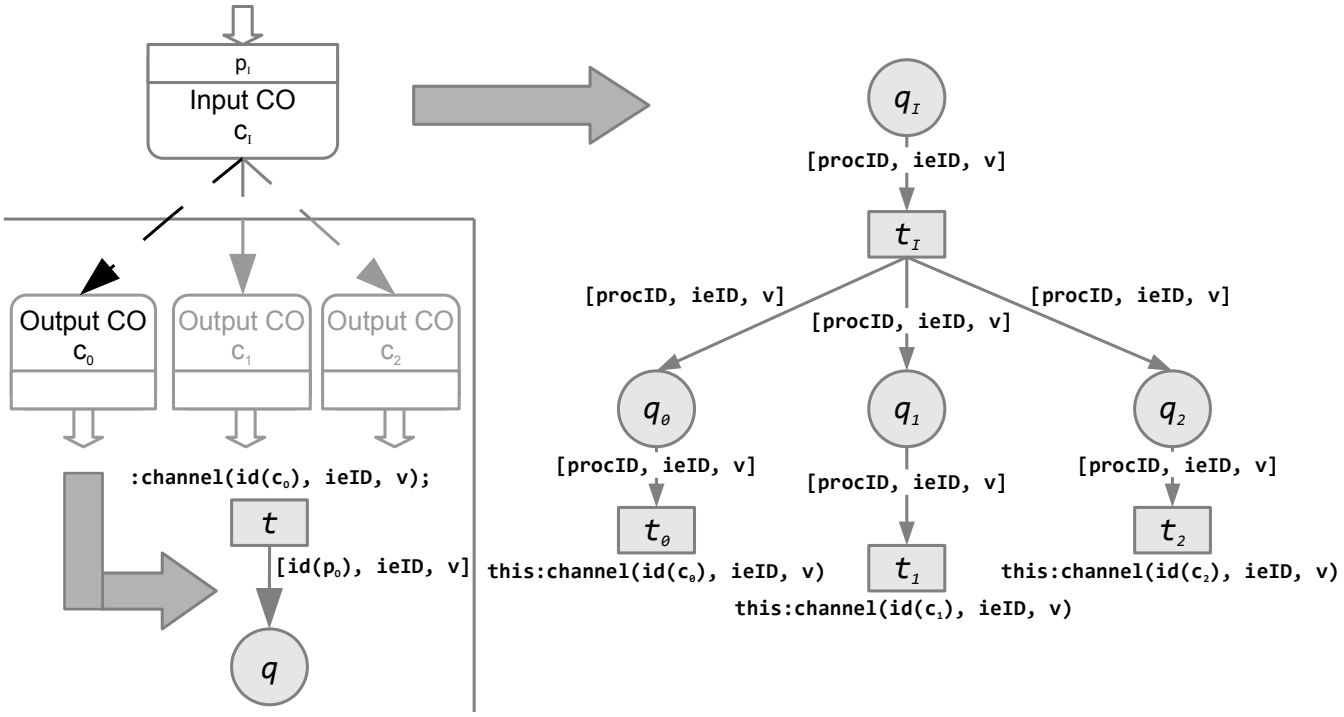

Figure 5. Transformation of channel operations into a reference net.

Subsequently, the transformation generates an AND like subnet for every group and thereby behaves like an XOR node between the groups. If all incoming processes of one group provide a data object each, the group's associated transition can fire, independent from other groups. Guard conditions control which data objects are redirected to the outgoing process.

OR(branch): Outgoing process groups are triggered according to the guard condition defined in the FILL graph. The assignment of the guard conditions to the group is defined by an arrow in the FILL graph's guard condition, as it is the case for all guard condition assignments for edges in the above cases of *AND* and *XOR* nodes.

3) Transformation of proxy nodes: Proxy nodes represent data connectors to and from interaction elements. Therefore, κ is a function relating a proxy node to its associated interaction element by a unique reference. This reference is used as specification of a channel name in case of an output proxy node (see Figure 6 left), such that an event can be uniquely redirected to the correct proxy node representation in the reference net. The callback function from the net to the physical representation and the associated interaction element is specified by a fixed channel name called `widgetCallback`. To identify the correct interaction element on the side of the physical representation, its identifier (given by κ) is passed as parameter to the channel (see Figure 6 right).

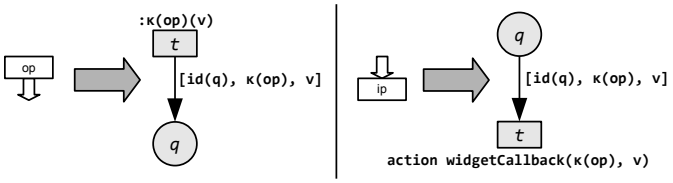

Figure 6. Transformation of proxy nodes into a reference net.

The section below will present a comprehensive example of the use of FILL and an associated transformation to reference nets. The example provides a deeper insight to the semantics of a FILL graph and how an associate reference net transformation looks like before Section III-D introduces the rewriting approach for interaction logic models.

C. FILL Example

Figure 9 shows a FILL graph that consists of two interaction processes; on the left an interaction process is shown using an XOR BPMN node, on the right a simple interaction process that triggers a system operation is shown. The latter process starts with an interaction-logic operation that only supports a single output port without any input port. Thus, the operation only emits data objects into the process but does not consume any. In this case, the operation called "ticker" sends a simple object into the interaction process and thereby triggers the following system operation called "getSV2Status". This system operation returns the associated value, here the status of SV2 that represents whether a steam valve of a simple nuclear power plant simulation is open (true) or closed (false). This value is then sent to an input proxy connected to an interaction element, such as a lamp widget that flashes green in case of a true value and red in case of a false value.

Before discussing the interaction processes in more detail, the simple nuclear power plant simulation will be briefly presented. In Figure 8, the process is shown. The nuclear power plant simulation consists of three main elements: the reactor, the condenser, and the turbine, which transfers steam into rotation energy that is further transferred into electrical energy

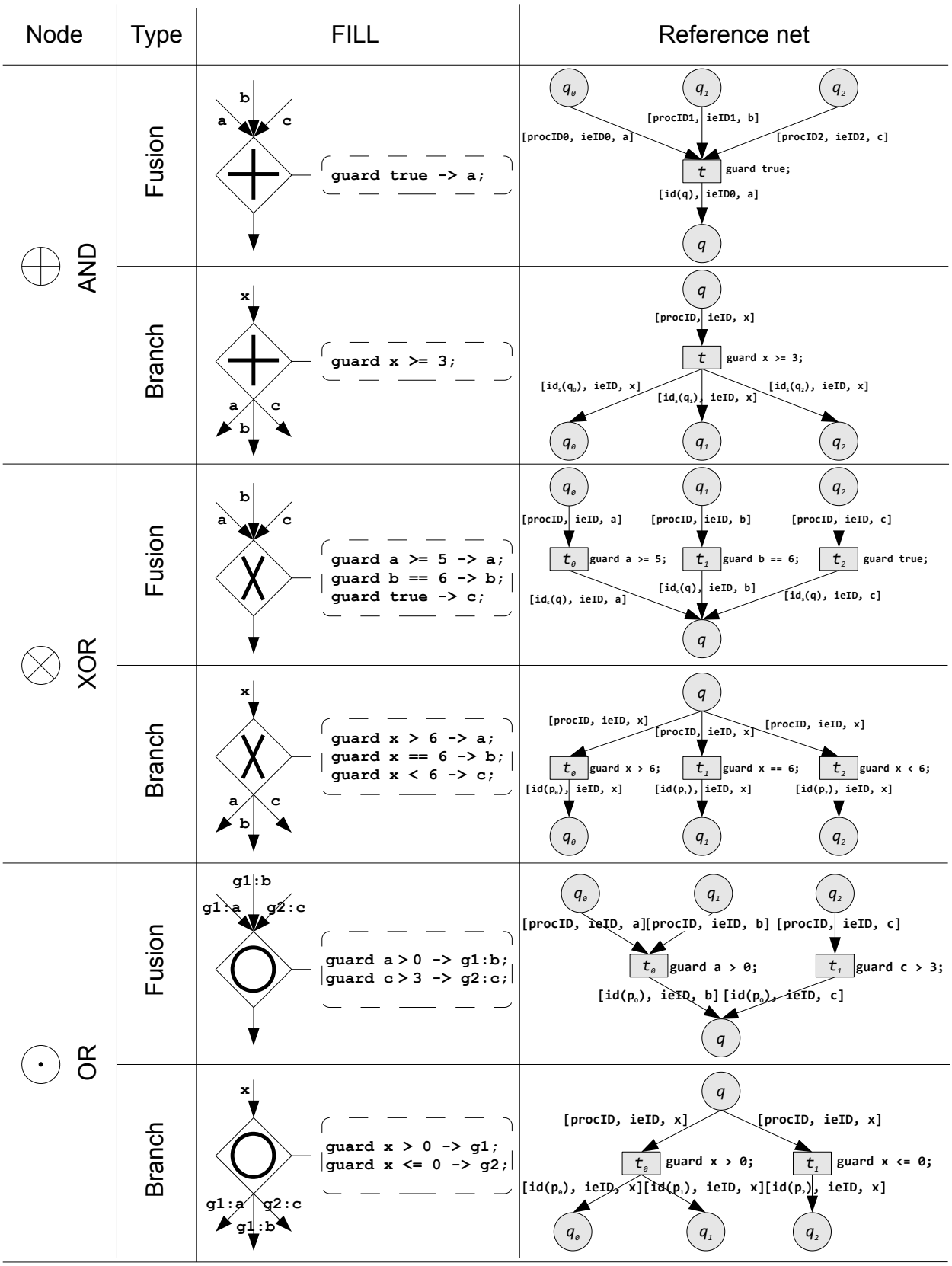

Figure 7. Transformation of BPMN nodes into a reference net.

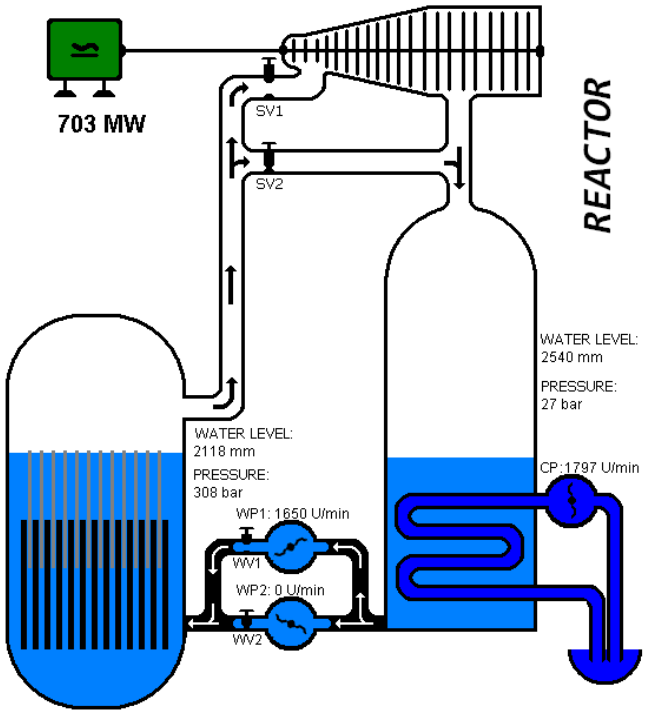

Figure 8. Feed water circuit of a simplified simulation of a steam water nuclear reactor called *REACTOR*.

through a connected generator. The condenser condenses the steam back into fluid water that is pumped by two pumps back into the reactor. The nuclear reaction generates thermal energy and is controlled by control rods, which can be pushed into or removed from the nuclear core. Removing increases the amount of thermal energy boiling the water and thereby generating steam. Various valves can be further used to control the way of the water and the steam.

The process on the left of Figure 9 also includes an XOR BPMN node branching the interaction process into two sub-processes. The whole process is triggered by an interaction element, like a button, sending an event object to the interaction process. Before the XOR node is triggered, the same system operation is triggered as in the left interaction process, such that the status of the valve is sent to the XOR node. The associated guard condition specifies that in case of a false value, the sub-process indicated with `a` will be triggered by the inputted value. In case of a true value, the sub-process `b` will be triggered. In both cases, an interaction logic that generates a Boolean value and emit this to the system operation "setSV2Status" is executed. Thus, if the current value of the steam valve is true, it will be changed to false and vice versa.

The result of the transformation to a reference net is shown in the middle of Figure 9. Gray boxes with different types of borders indicate which FILL element is transformed into which subgraph of the reference net. Furthermore, it can be seen how edges in the FILL graph are transformed into transitions, simply redirecting incoming data objects to their outgoing edges. `procIDs` are used to specify a certain interaction process throughout the reference net. This is of special interest in case of transformation of system operations, which have only one representation in the reference net. This has various reasons. System operations can influence or return (part of) the system state; these operations are state-full. Write-write race condition should be avoided in case of state-full software components and thereby only single representations of system operations exists. Please note that the presented example intends to give a deeper insight how a result looks like that the algorithm generates.

Therefore, procIDs are necessary to identify the correct reentering point after returning from a system operation. In Figure 9, this can be seen in case of the system operation "getSV2Status". Here, the relevant procIDs are marked with dashed ellipses.

Interaction-logic operations are in contrast to system operations transferred into multiple sub-nets in the reference net. For every used interaction-logic operation block in the FILL graph, a subgraph is created. In this case, the above discussed concurrent method calls could occur. Still, interaction-logic operations should not be state-full. Thus, the race conditions as described above will not occur. Additionally, pre-defined parameters, such as the parameter specifying which Boolean value should be generated is different between every use of the interaction-logic operation. In Figure 9, the operation "generateBooleanValue" is called once with the parameter `new Boolean(true)` and once with the parameter `new Boolean(false)`.

D. Formal Reconfiguration

Adaptive user interfaces offer great opportunities in human-computer interaction (what has been discussed in detail, above). Thus, formal user interface models should be enabled to be adaptive or even adaptable in a certain sense. This section introduces a formal reconfiguration concept that is able to adapt the presented graph-based user interface modeling approach.

Formal reconfiguration can be differentiated from redesign, where redesign refers to the change of the physical representation and reconfiguration specifies changes in the interaction logic of a user interface model. Interaction logic is modeled using FILL and then transformed to reference nets, not only for defining formal semantics but also for making FILL graphs executable. Thus, reconfiguration means changing reference nets, necessitating a method (a) that is able to change reference net models and (b) that is defined formally to prevent reconfigurations from being non-deterministic. Various graph transformations and rewriting approaches can be found in literature. Shürr and Westfechtel [38] identify three different types of graph rewriting systems. The logic-oriented approach uses predicate logic expression to define rules. This approach is not wide-spread due to its complex implementation. Another approach defines rules based on mathematical set theory, which is flexible and easily applied to various applications. Still, it has been shown that irregularities could occur applying set-theoretical rules to graph-based structures.

Finally, graph rewriting based on category theory has been chosen for reconfiguration according to various features. First of all, pushouts (see Definition 3) as part of category theory are well behaved regarding their application to graphs, especially the double-pushout (DPO) approach as has been discussed by Ehrig et al. [39]. The DPO approach specifies rules that

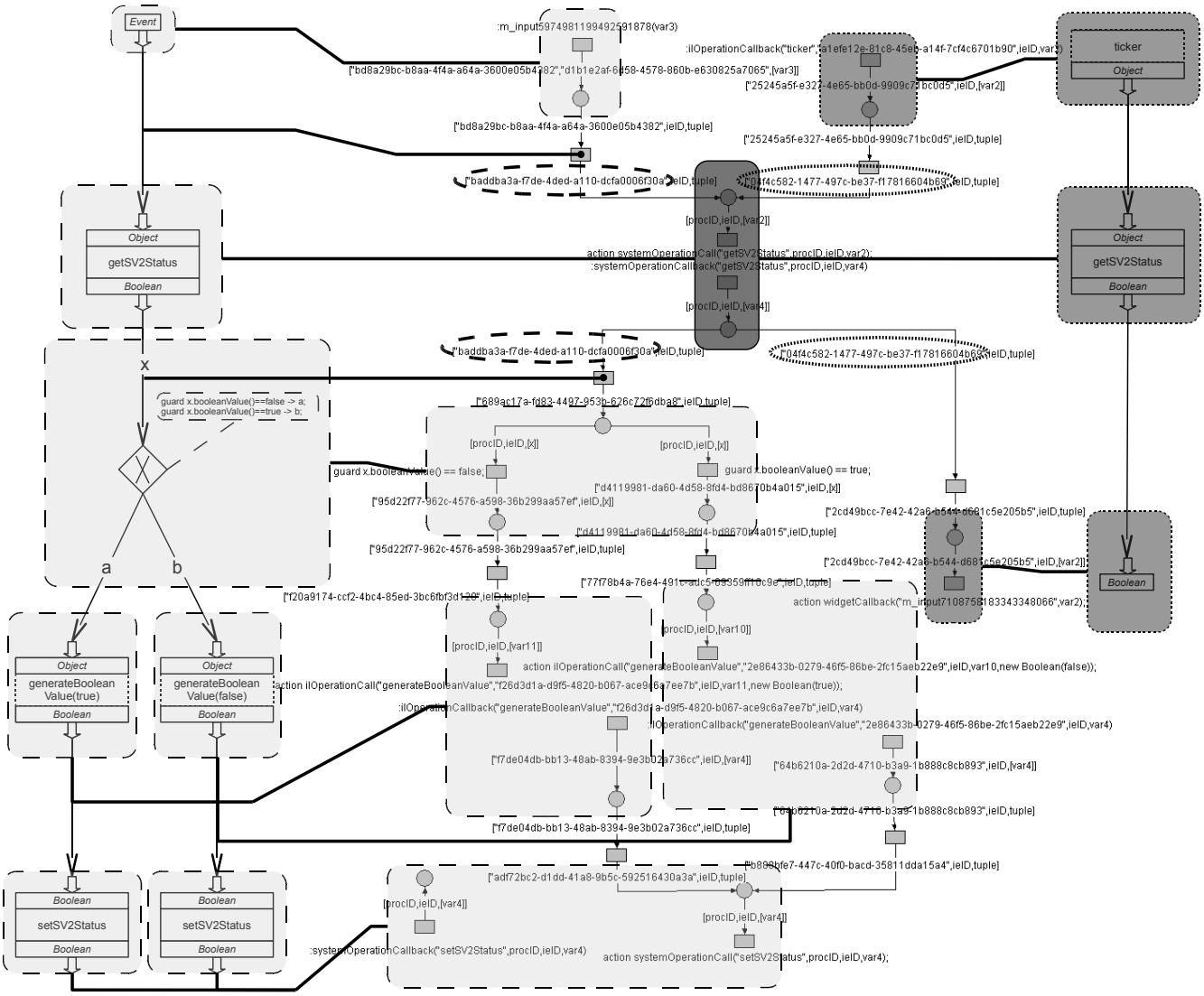

Figure 9. An example of a FILL graph with two interaction processes, showing their corresponding transformation to a reference net.

explicitly define which nodes and edges are deleted in a first step and then being added to the graph in a second. This is not true for the single-pushout (SPO) approach, which is implementation dependent or generates results that are probably not valid graphs [39]. Only to give a simple example, the SPO approach can afford dangling edges, which are edges having only a source or a destination, but not both. A further problem could be the implicit fusion of nodes, which could have negative implications to the rewriting of interaction logic. These aspects have been resolved in the DPO by making deletion and adding of nodes and edges explicit as well as defining a condition preventing rules from being valid if they produce dangling edges. The DPO approach definition will make this more precise, as given below.

A further argument supporting the use of the DPO approach for rewriting interaction logic is that it has been extended and discussed in context of Petri nets as discussed by Ehrig et al. in [40] and [41]. This work offers a solid basis for the reconfiguration of reference net-based interaction logic. Furthermore, the DPO approach (as well as the SPO) is able to change existing graphs, where graph grammars are production systems. Using graph grammars for reconfiguration would mean to change production rules instead of defining rules changing an existing graph. At a first glance, this seems to be less comfortable and counter intuitive. Finally, the Petri net-based DPO approach as described by Ehrig et al. [40] can be simply extended to colored Petri nets. Within certain boundaries, also the semantics of the inscription can be taken into account, as described in detail by Stückrath and Weyers [42].

As the SPO, the DPO is based on the category theory-based concept called pushouts. Assuming a fundamental understanding of category theory (otherwise consider, e.g., [43]), a pushout is defined as follows.

Definition 3: Given two arrows $f : A \to B$ and $g : A \to C$, the triple $(D, g^* : B \to D, f^* : C \to D)$ is called a *pushout*, D is called *pushout object* of (f, g), and it is true that

1) $g^* \circ f = f^* \circ g$, and

2) for all other objects E with the arrows $f' : C \to E$ and $g' : B \to E$ that fulfill the former constraint, there has to be an arrow $h : D \to E$ with $h \circ g^* = g'$ and $h \circ f^* = f'$.

The first condition specifies that it does not matter how A is mapped to D, that is via B or C. The second condition guarantees that D is unique, except isomorphism. Thus, defining (f, g) there is exactly one pushout (f^*, g^*, D) where D is the rewritten result, also called *pushout object*. In general, A and B are given defining the changes applied to C, the graph to be rewritten. Therefore, a rewriting rule can be specified as a tuple $r = (g, f, A, B)$, such that D is the rewritten result by calculating the pushout (object). This procedure is mainly applied in the SPO approach.

For the definition of the DPO approach, the pushout complement has to be defined first.

Definition 4: Given two arrows $f : A \to B$ and $g^* : B \to D$, the triple $(C, g : A \to C, f^* : C \to D)$ is called the *pushout complement* of (f, g^*) if (D, g^*, f^*) is a pushout of (f, g).

A DPO rule is then defined based on the definition of a *production* corresponding to the former discussion of pushouts in category theory.

Definition 5: A *matching* is a mapping $m : L \to G$; a *production* is a mapping $p : L \to R$, where L, R, and G are graphs. The corresponding mappings of m and p are defined as mapping $m^* : R \to H$ and $p^* : G \to H$, where H is also a graph.

Definition 6: A *DPO rule* s is a tuple $s = (m, (l, r), L, I, R)$ for the transformation of a graph G, with $l : I \to L$ and $r : I \to R$, which are two total homomorphisms representing the production of s; $m : L \to G$ is a total homomorphism matching L to graph G. L is called the *left side* of s, R is called the *right side* of s, and I is called an *interface graph*.

Given a rule s, in a first step the pushout complement C can be calculated using L, I, m, and l with a given graph G to be rewritten. In the DPO approach, this step deletes nodes and edges from G. In the second step, the pushout is calculated using I, R, and r applied to C resulting in the graph H. This step adds nodes and edges to C. In conclusion, the difference between L and I specifies the part deleted from G, where the difference between I and R defines those elements, which are added to C and finally to G. The result of applying s to G is the graph H as can be seen in Figure 10.

Nevertheless, the pushout complement is not in all cases unique or probably does not even exists. According to the latter, if the total homomorphisms l and m fulfill the *gluing condition*, the pushout complement will always exist. The gluing condition is defined as follows.

Definition 7: There are three graphs $I = (V_I, E_I, s_I, t_I)$, $L = (V_L, E_L, s_L, t_L)$, and $G = (V_G, E_G, s_G, t_G)$. Two graph homomorphisms $l : I \to L$ and $m : L \to G$ fulfill the *gluing condition* if the following assertions are true for both l and m, given as

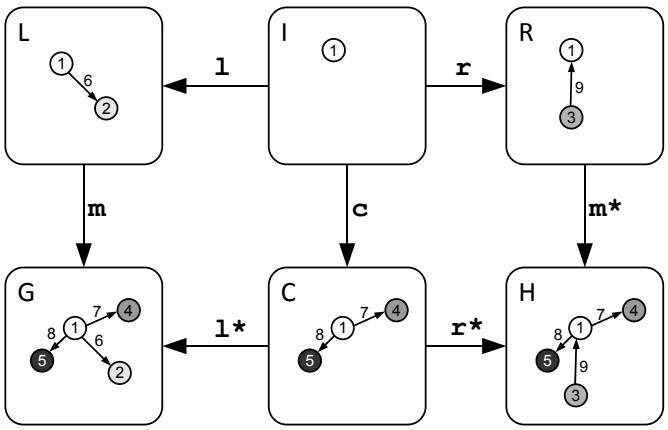

Figure 10. Exemplary DPO rule and its application to a graph G.

$$\nexists e \in (E_G \setminus m(E_L)) : s_G(e) \in m(V_L \setminus l(V_I)) \vee t_G(e) \in m(V_L \setminus l(V_I)), \tag{18}$$

and

$$\nexists x, y \in (V_L \cup E_L) : x \neq y \wedge m(x) = m(y) \wedge x \notin l(V_I \cup E_I). \tag{19}$$

Condition 18 is also called *dangling condition*. The homomorphism l of a DPO rule that defines, which nodes have to be deleted from a graph fulfills the *dangling condition* if it also defines which edges associated with the node will be removed. Thus, the dangling condition avoids *dangling edges*; a dangling edge is an edge that has only one node associated with it as its source or target. Condition 19 is called *identification condition*. The homomorphism m fulfills the identification condition if a node in G that should be deleted has no more than one preimage in L. However, if one node of G has more than one preimage in L defined by m and one of these has to be deleted, it is not defined whether the node will still exist in G or must be deleted. This confusion is avoided by the identification condition.

The problems of the SPO approach discussed above are mainly solved by the gluing condition being an integral part of the DPO approach. Nevertheless, the pushout complement is not unique but exists if the gluing condition is fulfilled. If l and m are injective, the pushout complement will be unique except isomorphism. This aspect is further discussed by Heumüller et al. [44] and in [7, p. 107].

The above definition of the DPO approach is only applied to simple graphs (cp. Figure 10). An extension of this approach to (simple PT) Petri nets has been discussed by Ehrig et al. in [40] and [45], and Weyers [7]. Nevertheless, rewriting of inscriptions has not been considered by these authors. Inscriptions extend basic Petri nets with further semantics, such as supporting complex data objects as tokens and the definition of guard conditions that extends the firing semantics of transitions. Still, rewriting interaction logic means rewriting reference nets, which are finally Petri nets with a specific inscription language that supports guard conditions and the definition of synchronous channels. Therefore, rewriting inscriptions has been discussed by Stückrath and Weyers in [42].

Finally, rule descriptions have to be serialized in a computer readable form. This is necessary for the implementation-side use of rewriting rules using the algorithmic rule generation concept presented in the next section. Therefore, an existing XML-based description language for Petri nets (PNML [46]) has been used with an extension to structure and describe the rule specific elements, such as indicating L as `deleteNet`, I as `interface`, and R as `insertNet`. The `mapping` node specifies m as a set of `mappingElement`s, which are representations of tuples of XML ids. An example showing a rule in XML format can be seen in the XML snippet included in Appendix A.

E. The UIEditor

The above sections introduced a formal modeling and rewriting approach for the creation and reconfiguration of user interfaces, especially for the description of interaction logic. This approach is accompanied with a tool implementation called UIEditor, which offers capabilities for the creation/modeling, execution, and reconfiguration of user interface models. Figure 11 illustrates a general workflow implemented by the UIEditor that characterizes the creation and usage of user interface models based on our approach. In the *creation phase*, the user interface model is interactively created by an expert using the visual editor for editing the physical representation and the interaction logic using FILL. Figure 12 shows the UIEditor in creation mode with two opened visual editors. On the right, the drag-and-drop editor for the creation of the physical representation can be seen. On the left, the editor for creating the interaction logic using FILL is shown. Generally, FILL nodes and interaction elements are added to the canvases via drag-and-drop. In the case of FILL nodes, each node is connected via edges from output to input ports in an interactive fashion. Parameters of interaction elements can be specified through context menus and associated dialogs.

For the execution of user interface models, the UIEditor implements the transformation algorithm discussed in Section III-B, which is encapsulated in the creation of simulation files. These files define the connection of reference net-based interaction logic models, the description of the physical representation, the associated system to be controlled, and some further parameters like logging. After finishing the transformation, the UIEditor changes into simulation mode loading the simulation file and starting the execution of the user interface. In *simulation mode*, the UIEditor shows only the physical representation of the user interface model and starts the underlying controlled system. The interaction logic is loaded to Renew. The UIEditor handles the communication between reference net simulation, the physical representation, and the system to be controlled. Here, the user is able to interact with the system using the modeled and executed user interface.

If the user stops the simulation (by pressing the stop button), the UIEditor changes into *reconfiguration mode*. Here, the physical representation is shown to the user with a set of buttons related to the execution of reconfiguration rules. Various reconfiguration operations are supported, such as parallelization of input operations or discretization of interaction elements. In the first case, the user selects n different interaction elements and applies the parallelization operation to it. A new

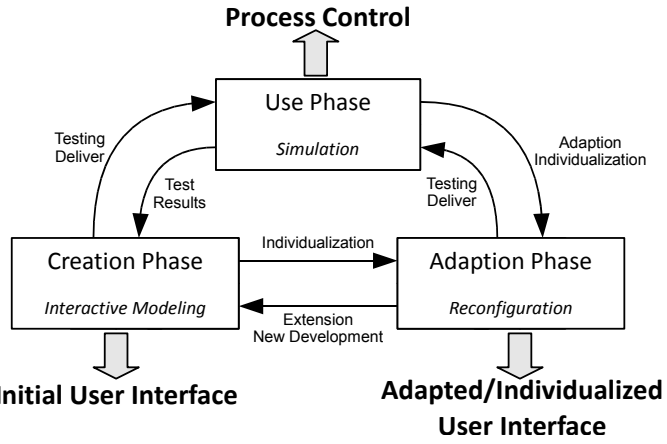

Figure 11. General workflow for using the UIEditor: creating user interface models, running these through simulation, and reconfigure these using an interactive tool.

button is generated that executes all operations in parallel that were previously connected to the selected interaction elements. Discretization is a reconfiguration operation that generates a button, which sends a predefined value to an interaction process that has been previously connected to a continuous interaction element, such as a slider. In the current implementation, these operations generate DPO rules for changing the underlying interaction logic based on an algorithmic and hard-coded implementation. This makes the reconfiguration mechanism inflexible and hard to extend. Therefore, the next section will introduce a more flexible concept for describing and instantiating rules in a more flexible and extensible way using abstract description concepts and formats.

IV. Rule Generation

The current implementation of reconfiguration in the UIEditor is exclusively user-driven and based on pre-defined algorithmic implementations for rule generation making the whole reconfiguration approach inflexible and hard to extend. Therefore, this section describes an extension based on the modeling and reconfiguration formalism introduced above (see Section III) according to (a) a generic and simple-to-use description of rule classes and (b) the algorithmic implementation of rule generation based on these classes. In general, the approach aims at the generation of DPO rules applied to reference net-based interaction logic as well as basic redesign rules applied to the physical representation, where the generation of DPO rules will be the focus of the discussion. The rule generation enables reconfiguration and redesign to be applied automatically as well as by the basic approach of user triggered reconfiguration, as discussed above. The whole approach is embedded into a rule generation process, which will be discussed in the following section. This process identifies necessary steps and algorithmic implementations for the intended rule generation, which will be presented in more detail in Section IV-B and IV-C. Finally, the approach will be discussed with an example of adaptive automation in Section V relating to the previous work discussed in [1].

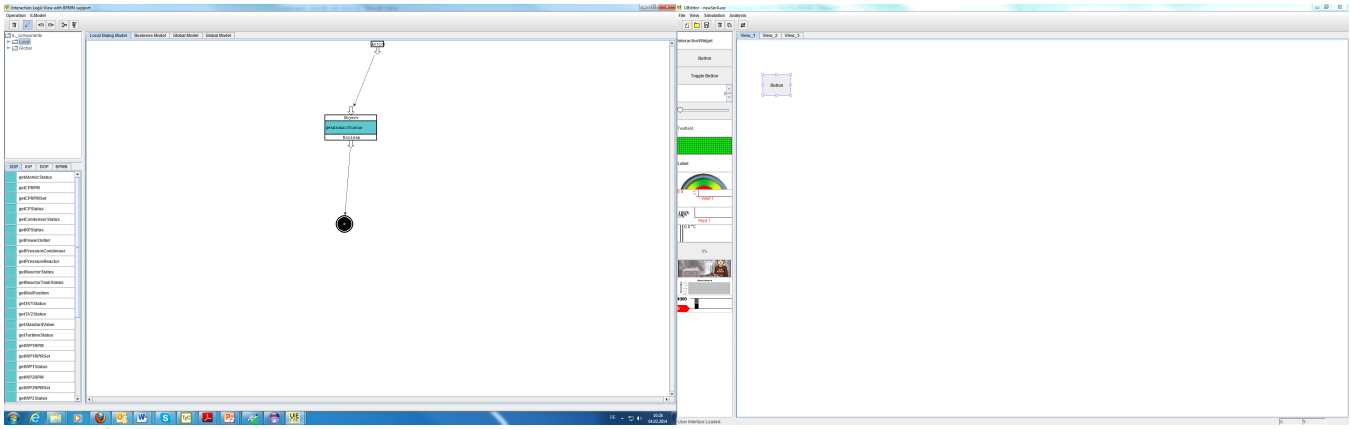

Figure 12. The UIEditor in creation mode: left, the interactive editor for modeling FILL graphs, right, the interactive drag-and-drop editor for modeling the physical representation.

A. Rule Generation Process

The rule generation process is illustrated in Figure 13. It specifies the main steps for selecting necessary information and data for the rule generation, certain validation and decision steps, and finally the rule generation and the rule application to the interaction logic and the physical representation of the user interface model.

First, the process has to be triggered, either by the user who interactively selects certain interaction elements and a rule to be applied or by the system that decides to adapt the user interface based on sensory data and certain domain knowledge. Still, this part of the reconfiguration process is not part of the approach discussed here regarding its domain and use case dependent implementation. However, this trigger operation ends up in a call of a user interface reconfiguration (indicated by ① in Figure 13) and in the gathering of input data, which is essential for further steps in the process. In step ①, a matching rule class is selected from a data base according to the information and data provided by the triggering instance. In step ②, the rule class gets validated based on various aspects. One major validation step here is the analysis of input data according to completeness and correct matching to the selected rule class, as well as the evaluation of the class' instantiation precondition. Therefore, rule classes specify preconditions according to data that has to be provided for a successful instantiation.

If the rule class is validated to be applicable, it is redirected to step ③ generating the rule. Otherwise, the process terminates. The rule generation step will be discussed in more detail in Section IV-B below. In general, the rule generation takes a rule skeleton as input and tries to create a set of rules using the genuine (interaction logic) net and the input parameters provided by the triggering instance. If the set is empty, no matching subnets in the genuine interaction logic could be found and the process terminates. If the set holds more than one rule, a rule has to be selected in step ④ using a certain heuristic. One simple approach would be to select one rule randomly or the first that is successfully generated. Alternatively, it could be also possible to select the rule that has the greatest impact on the rewritten net. Nevertheless, the used heuristic has to be specified in context of the rule class description, as will be further described in Section IV-C. After the rule has been applied to the genuine net in step ⑤ according to the DPO-based rewriting approach (discussed above in Section III-D), the rule class is checked according to necessary changes in the physical representation. If the physical representation has to be redesigned, the specified changes are applied to the user interface in step ⑥. Finally, the process terminates.

B. Rule Skeleton

Step ③ shown in Figure 13 generates rule instances from rule skeletons using input data and other parameters. This section will specify the structure of rule skeletons and the generation algorithm extracting rule instances from a given skeleton, which involves linking of resources into the skeleton. This is necessary to extract certain structures from the net to be rewritten in a second step. Rule skeletons are encapsulated into rule classes, which provide further information regarding validation, necessary data for instantiation, and directives for possible redesigns being applied to the physical representation of a user interface model. Rule classes will be discussed in more detail in Section IV-C.

A skeleton specifies three special types of graphs L, I, and R that represent a DPO rule, as well as the functions l and r (cf. Figure 10). Figure 14 shows a simple example of a rule skeleton. Here, a new interaction element (a slider)

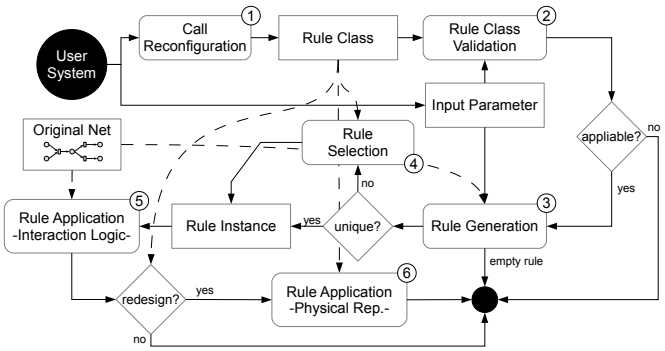

Figure 13. The rule generation process.

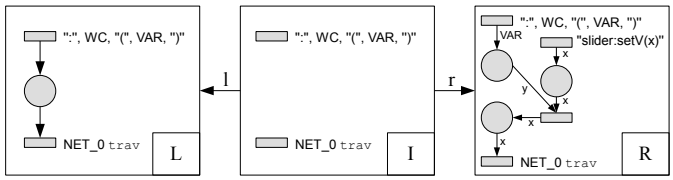

Figure 14. Simple example of a rule skeleton.

is added that is combined with an existing one, such that the existing element has to emit an event before the slider's value is redirected into the existing interaction process. The underlying system could, again, be the simple simulation of a nuclear reactor, as has been mentioned above (see Section III-C). In this case, the slider can be used to set the rounds per minutes or speed of a water pump in the system and then in a second step to set this value to the system by clicking the button.

1) Graph and text grammars: Rule skeletons are mixtures of graph and string grammars as well as control structures, such as loops or alternatives. Graph grammars are "... similar to a string grammar in the sense that the grammar consists of finite sets of labels for nodes and edges, an axiom, i.e., an initial graph, and a finite set of productions" [47, p. 120]. Graph grammars have been shortly presented above when discussing various rewriting systems (see Section III-D). In this context, graph grammars were not suitable because they are defined by rules that generate graphs and as opposed to change existing graph structures. In context of rule generation, grammatical descriptions are instead suitable because DPO rules need to be *generated* (and not rewritten) and thereby the graphs L, I, and R in particular.

In case of graph grammars, nodes inscribed with nonterminals are substituted during the generation by graphs or specific inscriptions, for instance extracted from the original graph through graph traversing or by given graph structures offered as initial parameter. Furthermore, inscriptions gets substituted by additional data or inscriptions extracted from the original data and the genuine net. Which data source is used for a substitution of nonterminals in the skeleton is specified in the rule class description or results from the expansion algorithm described below. The following list specifies which types of nonterminals can be used in rule skeletons, where EBNF refers to the Extended Backus-Naur Form, a special type of language for the definition of textual grammars:

- *EBNF like nonterminal symbols:* These nonterminals are used in inscriptions to be replaced by matching the associated node to a node in the genuine net (the net to be rewritten by the resulting rule) or by matching it to predefined parameters as has been inputted or specified by the rule class. In general, inscriptions are specified using EBNF syntax. Nonterminals are printed in capital letters only, such as `WC` in Figure 14.
- *Net nonterminal symbols:* Nodes inscribed by `Net` nonterminals getting replaced by (a) a subnet extracted from the genuine net or (b) by a predefined net given by the rule class or as additional input data. In case (a), the nonterminal in the skeleton is extended by a keyword specifying how the subnet has to be extracted from the genuine net. The example shown in Figure 14 (`NET_0 trav`) will be replaced by a traversed subnet of the genuine net. In case (b), the rule class has to specify by which net the nonterminal has to be replaced. In general, a net nonterminal is indicated by the keyword `NET_X`, where `X` specifies a further identifier of the nonterminal making it usable several times in the same skeleton.

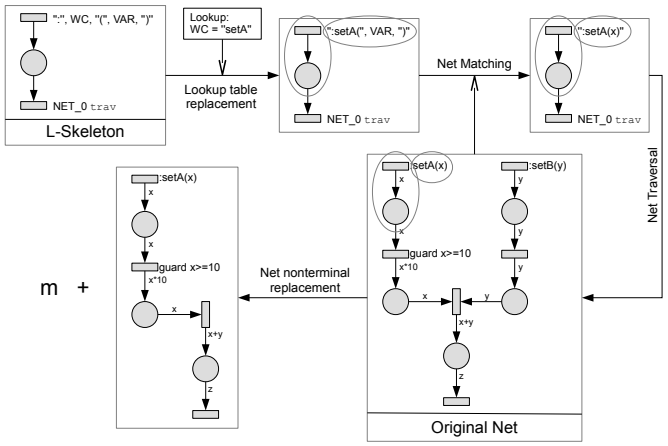

Figure 15. Example for nonterminal replacement by applying the REP algorithm.

For the instantiation of a rule skeleton, first nonterminals of the left side of the rule skeleton are replaced and the matching function m that is an essential part of the rule instance is derived by matching the left side to the genuine net. In a second step, replacements in graphs I and R are made according to replacements made in the first step and using given data and information stored in a lookup table. In Figure 15, a sample graph extracted from the rule skeleton (as shown in Figure 14), which is using both types of nonterminals can be seen. Before discussing the example, the replacement algorithm, which is capable of collecting the replacements for nonterminals and apply the replacements to the skeleton, will be described in more detail.

Replacement Algorithm (REP)

1) *Lookup table generation:* This table is generated from input parameters and values specified in the rule class. It can contain values of various types including nets. Keys are derived from the rule class and should match names of nonterminals in the skeleton. An empty or invalid lookup table is prevented by the rule class validation step executed before the replacement (Figure 13 ②).
2) *Lookup table replacement:* All nonterminals with matchings in the lookup table are replaced in the skeleton's nets.
3) *Net matching:* Based on the partially replaced skeleton and especially of the partially replaced net L, a possible matching in the genuine net is identified to
   a) *replace nonterminals* that have not been replaced using the lookup table and
   b) find entering points/nodes in the genuine net for a *net traversal*.

4) *Net traversal:* The genuine net is traversed according to the previously found matching(s) and due to the specified traversal method (as specified in the rule class), which will be described in more detail below.
5) *Net nonterminal replacement:* Net nonterminals are replaced by the previously derived subnet(s).
6) *Completion of lookup table:* The lookup table gets extended with the derived subnets from net matching and traversal bound to the nonterminal names used in the skeleton.
7) *Initiate redesign:* The redesign as defined in the rule class is initiated, such that all nonterminals are replaced in the redesign and the redesign is applied to the physical representation.

Net traversal is basically implemented by a simple algorithm. It traverses the genuine reference net simply by following the directed edges through the net from a pre-defined starting node (derived from a matching of the rule skeleton's nets to the genuine net). Guard conditions and further semantic information are ignored for the traversal because the traversal aims at extracting a certain sub-structure of the net without considering the token play during runtime. If the traversal algorithm hits a transition representing an operation node, the inscription is interpreted regarding the identification of the corresponding recall transition. Here, the relevant ID is extracted from the inscription and tried to be matched to other transitions' inscriptions. If the algorithm cannot find a matching transition, it terminates. A second termination condition of the traversal is that the algorithm hits a node in the net that has no further edges leaving it.

Net matching and net traversal are further used to derive the matching function m necessary for the final rule definition. Note that m specifies the subnet of the genuine net that is rewritten by the rule. Furthermore, the net matching is possibly not unique leading to 0 to n resulting rules from this step in the algorithm. In the case of $n > 1$ matchings, n rule instances result from the process (Figure 13).

Figure 15 shows the application of the REP algorithm onto the left side of the rule skeleton, as defined in Figure 14. Assumed that `WC` is predefined by given input data, such that `WC` is defined as the string `setA`, the initial lookup table is filled with this information. Therefore, in the first step `WC` is replaced by `setA` resulting in the first intermediate result. In the next step, the genuine net is matched to the intermediate result. Thereby, the nonterminal `VAR` can be replaced by `x`. Subsequently, the genuine net is traversed given the initial node, in this case the transition is inscribed with `NET_0 trav`. Finally, the result of the traversal is added to the skeleton.

The whole rule instance is derived by further application of the lookup table replacement step of the REP algorithm to all nonterminals. According to the algorithm's last step, the traversed subnets are part of the lookup table and thereby can also be replaced in I and L. Thus, the lookup table should contain all necessary replacement elements due to the rule class validation and the previously applied net traversal.

2) Boxing: Up to this point, only graph grammar and EBNF-like replacements of nonterminals applied to graphs were used. For more complex rules, for instance, rules involving a number of interaction elements that are not fixed in advance, further structures are necessary extending the current modeling approach of rule skeletons. Therefore, the basic skeleton description is extended by *boxes* that represent *loops* or *alternatives*, which are parameterized during runtime as specified in the rule class. Thus, from this boxed description of a rule skeleton, a simple nonterminal-based representation is derived algorithmically before the REP algorithm generates the final rule. This extraction algorithm is composed of recursive calls of loop box and alternative box extractions following the nesting of boxes of both types. Before discussing the extraction algorithm in more detail, the semantics of loop and alternative boxes will be discussed in detail.

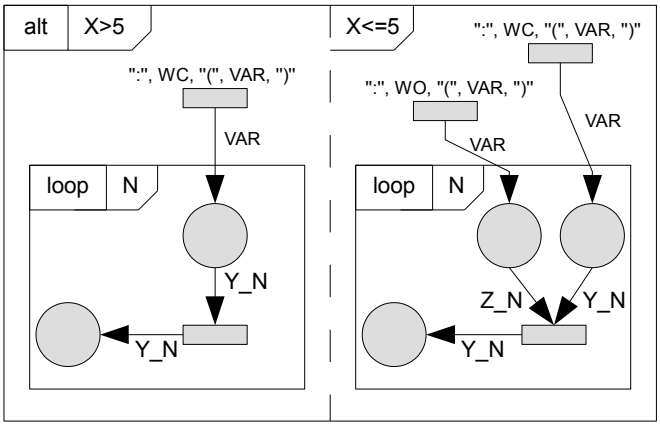

Figure 16. Example of using boxes in rule skeletons: a combination of alternative and loop boxes.

Loop box: A graph defined in a loop box gets replicated as many times as defined. Therefore, a loop box gets parameterized by the number of copies that should be created. Figure 17 shows a loop box indicated by the keyword `loop` followed by the parameter `N`. This parameter can also be used as nonterminal in the graph. In Figure 17, the nonterminal `WC_N` represents for a call method dedicated to an individual interaction element. The iteration counter `N` added to the nonterminal `WC` specifies that every created copy of the given net has to be matched to a method name from an individual interaction element. The matching has to be specified in the rule class and generated during the generation of the lookup table in the REP algorithm.

Alternative box: This box defines two different graphs to be selected in the extraction phase according to the specified condition. This condition is evaluated in an if-then-else fashion. If the condition is evaluated to true, the graph in the left box is selected for further extraction, otherwise the right graph gets selected.

Boxes can be used in a hierarchical fashion, such that boxes of different type can be nested into one another. Boxes can also be used in parallel, such that one alternative box holds a graph that uses two loops on the same hierarchical level. In general, it is possible to box subgraphs, as it can be seen in Figure 17. Regarding the use of boxes in the specification of rule skeletons, these boxes have to be extracted before the replacement algorithm can be applied to a plain rule skeleton. The following algorithm is applied to boxed rule skeletons to

retrieve a plain rule skeleton for input into the replacement the REP algorithm. The extraction algorithm is mainly based on two steps: the first step resolves the nesting of the boxes and the second step applies the extraction to the boxed rule skeleton. The algorithm can be specified as follows:

Extraction Algorithm (EXT)

1) *Create Box Tree:* Starting on the highest level defining the root of the tree (which could be seen as the left, interface, or right side of the rule), all boxes on the next lower level are identified. For every box, one child will be created. This procedure is repeated until no more boxes can be identified on the next lower level (i.e., the lowest level has been reached in a subtree). If the currently selected node references an alternative box, the condition of the box gets evaluated before the children are inspected further, such that the selected subgraph can be inspected without creating unnecessary subtrees that would not be interpreted in the following step according to the evaluation of alternatives.
2) *Extraction:* To extract the final graph, the box tree is traversed in post-order. Each time a root node of a subtree gets selected by the traversal, the corresponding box is 'executed'. In case of an alternative box, nothing happens because it has already been evaluated in the box tree creation, before. In case of a loop box, the 'copy' operation is performed as often as specified. After finishing the box execution, the tree traversal process is continued until the box tree traversal ends in the root node.

In Figure 17, an example of an application of the extraction algorithm can be seen. To show the general process in all details, we decided not to discuss this example in context of the previous introduced simulation of a nuclear power plant (see Section III-C). The depicted rule stub is comprised of an alternative box and various loop boxes. Depending on the value `X`, one or two loop boxes have to be extracted. In the box tree creation step, the alternative box is inspected as the box on the highest level of the skeleton's box hierarchy. In this example, `X` is set to `3`. Thus, the right graph is selected for further inspection and a node is added to the box tree. Furthermore, the alternative box is removed from the rule skeleton as preprocessing of the extraction step. In the next step, the two loop boxes get inspected. For each box, a node is created in the tree each referencing one of the boxes. Next, the create box tree step terminates because no more boxes are encapsulated in the loop boxes.

The first extraction step as shown in Figure 17 starts with the traversal of the box tree (in post-order), first selecting the left loop box for extraction. The result of the first extraction step is shown on the lower left side of Figure 17. `N` is set to `2` resulting in two copies of the subgraph as defined in the rule skeleton. It can also be seen that the box crossing edges are duplicated in this case. The extraction of the second loop box is shown in the lower right corner of Figure 17, following the same extraction operation. The next node selected in the box node tree is the node representing the alternative node, which has been previously removed in the box tree creating step. The next and last node selected is the root node causing the EXT algorithm to terminate.

The EXT algorithm is applied to all graphs of the rule skeleton before the left side is inputted into the REP algorithm for nonterminal replacement. After finishing the EXT and REP algorithm, the rule has been generated. Before discussing a more complex example that implements adaptive automation using this approach, the rule class description has to be further specified, using XML as described below.

C. Rule Class Description

The rule class description, as briefly discussed above, specifies which data has to be provided to the REP and EXT algorithms to finally create the application specific rule. Beside the rule skeleton and the needed data, the rule class contains a description of necessary changes of the physical representation. At a glance, the following information is specified in a rule class:

- Metadata
- Selection heuristics and use of traversal algorithm
- Instantiation precondition
- Nonterminal declaration and definition
- Box parameter specification
- Rule skeleton
- Redesign of the physical representation

Metadata mainly specifies information regarding the sort of rule class, how it gets instantiated (interactively or system-side), its name, and some human readable description. The net traversal parameter specifies how the nets are traversed for rule skeleton instantiation. Currently, only the `standard` algorithm is implemented, as discussed above. The selection heuristic specifies how a certain rule is selected from a set of rules resulting from the instantiation of a rule skeleton. Currently, only the `first` strategy is available, simply selecting the first successfully generated rule. Further, a set of instantiation preconditions can be specified. Therefore, variables can be defined, which are used in these conditions in a second step. The variables are then matched against input data during runtime, similar to nonterminal symbols.

The declaration and definition of nonterminal symbols mainly specifies how values for the REP algorithm are derived or how certain nonterminals are associated to specific values. Values specifying box parameters are necessary to evaluate loop or alternative boxes in a rule skeleton. All specifications either define specific values (constants) or define how values will be derived; through user interaction or system side data input. Using concepts like RDF, the data specification in the rule class can be defined even more flexible. In Section V-B, this will be discussed in more detail with an example. Declaration of nonterminals and box parameters will be defined as shown in the XML snippet given in Appendix B.

In this context, RDF is used to specify nonterminal's datatype declarations and the specification of box parameters used in the associated rule skeleton. This makes an implementation independent description of rule classes possible, such that the concept is not restricted to be used with its initial implementation. For instance, the nonterminal `WO` (see

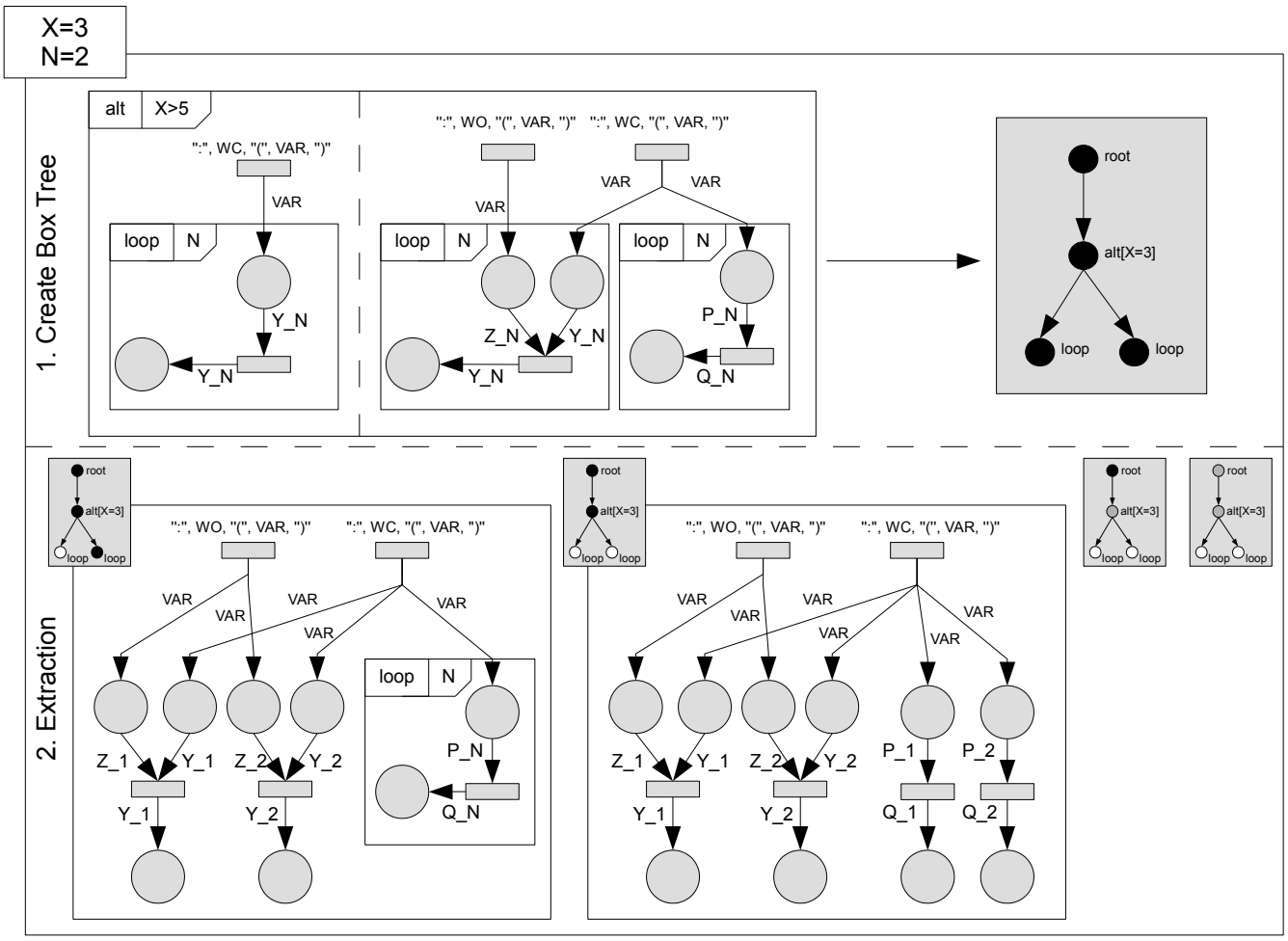

Figure 17. The application of the EXT algorithm, which extracts a boxed graph skeleton.

Appendix B) represents an interactively selected widget as can be identified by the specified datatype `widgetInteract`. `WC` represents a newly added widget, which needs an extension of the physical representation. This aspect will be further discussed below. Another example is the box parameter `X`, which is also inputted interactively as integer value (specified as `intInteract` datatype). Furthermore, a description has been specified to be shown to the user explaining what sort of parameter `X` is.

The generated rules are applied using the UIEditor implementation based on the DPO approach. This implementation uses a description of the DPO rule, where R, I, and L are defined as PNML-based graph specification. PNML (Petri Net Markup Language) is a markup language for specifying Petri nets of various types [48]. To stay consistent with this implementation, the description of rule skeletons will be based on this description concept simply extended by two types of XML nodes specifying loop or alternative boxes. The XML stub given in Appendix C shows how the rule skeleton in Figure 17 is expressed in the extended PNML format.

This extended net description is embedded into a rule description as discussed above (see Section III-D). Thus, three rule skeletons are specified with an additional specification of a mapping function.

In addition to changes of interaction logic, changes of the physical representation can occur, as it is the case for the nonterminal `WC` in Figure 14. In this case, a button should be added, which is connected to the interaction logic using the replacement of the `WC` nonterminal as reference. The REP algorithm is capable of creating a new interaction element as described in the rule class, adding it to the physical representation, and deriving the relevant reference from it to replace the relevant nonterminal in the rule skeleton. The redesign element for the rule class is described by the following XML snippet in Figure 18.

In this first version, only simple changes to the design of the physical representation are added: `newWidget` adds a specified widget to the physical representation and

```
<rc:redesign>
  <rc:newWidget reference="WC"
     widgetType=
       "http://uieditor.org/widgets/button"
     method=
       "http://uieditor.org/widgets/actionEvent"/>
  <rc:deleteWidget reference="WO"/>
</rc:redesign>
```

Figure 18. XML snippet of a redesign as specified in a rule class.

`deleteWidget` removes an existing widget from the physical representation. The interpretation system has to decide where to add the widget and in what initial size. In context of redesign, various extensions could be made, such as changing existing interaction elements regarding their outward appearance or their specific functionality. Still, this sort of change needs a more specific description concept of the physical representation and its redesign. The current version of the UIEditor uses a proprietary format for describing the physical representation. As future work, the use of UIML or UsiXML is planned to make the description of the physical representation more flexible and interchangeable between different (hardware) platforms.

Finally, an entire rule class is specified according to the example given in Appendix D. The following section will introduce and discuss an entire example using the rule generation process including the formalization of rule skeletons and rule classes based on XML as well as the application of the REP and EXT algorithms. The example presents a concept of adaptive automation as presented in [1]. It shows the usecase of a water pump as part of the nuclear power plant simulation scenario introduced above.

V. Adaptive Automation

Before presenting an example for adaptive automation in detail, the main motivation for implementing adaptive automation concepts will be discussed. The basis for this argumentation is that research in cognitive psychology has revealed important consequences of automation with respect to the human operator's workload in monitoring and control of technical processes, especially in critical, non-standard situations, as has been described in [49] and [50]. High workload is closely related to error rate, as well as to factors that influence the error rate in human-machine interaction, such as motivation, well-being, or situation awareness, as has been described in [51] and [52]. As has been discussed above, adaptive user interfaces are capable of reducing complexity in the interaction with technical systems [3]. Thus, it seems obvious to adapt user interfaces in order to suit particular users' needs and to introduce into the adaption process the degree of automation as an important parameter influencing human factors in human-machine interaction [53]. Here, the degree of automation defines whether the user has more or less control over the process, which system information in a critical situation is provided, or how the granularity of input operations is defined.

Therefore, adaptive automation will be discussed along the running example of a simplified nuclear power plant simulation. In this context, it is assumed that the mental workload of a reactor operator is measured for triggering and instantiating a user interface reconfiguration based on the rule generation concept discussed above and the UIEditor implementation with its associated formal user interface modeling approach. Especially in context of automated systems, the degree of automation is associated with a potential increase of mental workload and thereby is an indicator whether the degree of automation is too high or too low and whether it should be adapted or not. Weert [54] describes how mental workload can be measured based on different physiological factors, such as heartbeat rate, facial expression, perspiration, or eye blink rate. Out of these factors, pupillometry has been identified as a promising measurement tool for workload, especially in context of adaptive automation to increase human performance [55]. This gives an idea of how mental workload could be measured in the scenario presented above.

The section below discusses the use of the previously introduced approach for implementing a simple adaptive user interface, which is capable to adapt the degree of automation according to measured mental workload.

A. Example for Adaptive Automation

For making the degree of automation adaptable through a formal adaptive user interface model, it is assumed that the automation concept is fully accessible through an external formal model that matches the underlying concept of formal user interface modeling; therefore, the automation model should employ a reference net-based representation. If this assumption holds, the automation process can be also introduced into a rule skeleton and thereby can be introduced into the interaction logic through the rule generation and application process. Thereby, automation can be understood as formal abstraction of interaction processes between the human user and any given system that has been technically implemented. Thus, in our sense, automation is part of the interaction logic and simultaneously defines the degree of automation as visible from the user's perspective.

The automation of steering a water pump will be used as use case in the presentation, below. This use case of adaptive automation will be discussed according to a discrete and recurrent process of two operations: increase *(inc)* and decrease *(dec)* of the rounds per minute of a water pump. Here, it should be assumed that these operations have to be executed in an iterative fashion, such as the process shown in Figure 19. Thus, the process increases and decreases the rounds per minute (rpm) iteratively. According to the former assumption, this process can be introduced into reference net-based interaction logic as indicated by the bold arrow in Figure 19. The automation of this process can then be started by the user pressing the newly added "Start" button after the reconfiguration and redesign indicated by the bold arrow to the existing user interface. From this point on, the user is only able to monitor the system's state by observing the tachometer-like output widget, showing the pump's current rpm being controlled. Thus, using the reconfigured interface (indicated by **(I)** in Figure 19), she is not able to follow the operations that are automatically executed by the interaction logic.

As Parasuraman describes in [49] and [50] that workload increases during critical situations because the user has to understand the system's current situation, as well as how the automated control processes are reacting to the situation. The user has to gain insight into the automated process, resulting in an increase of mental workload, sometimes dramatically. This problem occurs in the example after the the first step of reconfiguration and redesign has been applied to the initial user interface. The user has no insights to the automated process, except that it is running. To adapt automation to this situation, the user interface **(I)** can be reconfigured (see Figure 19 **(II)**), by adding more interaction elements providing deeper insights into the automated process. Two lamps are added to

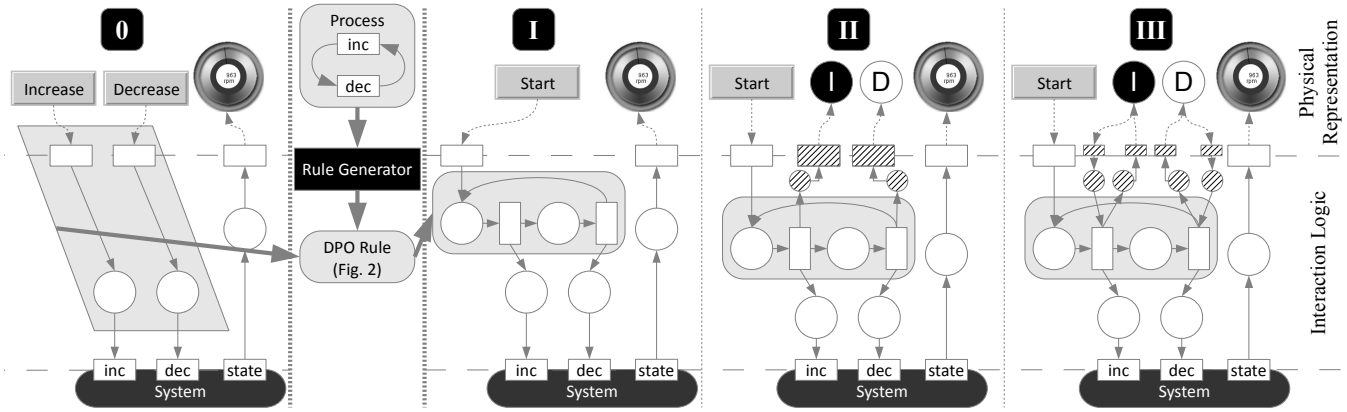

Figure 19. An example of how to change the degree of automation through reconfiguration of interaction logic, using rule generation.

the physical representation accompanied by an extension of the interaction logic, now showing which operation is executed at any given moment. Thus, the user is now able to see when the automation increases or decreases the speed of the pump. This makes interaction more finely grained and transparent to the user. A further reconfiguration extends the first by changing the simple lamps into buttons (see Figure 19 **(III)**), where the user is now able to control the automated process and thus gains more control over the still automated control process of the pump. Another possibility would be to remove the automation from the interaction logic and give all control back to the user without restriction or even, contrarily, to reduce the interaction and fully automate the process. The first would result in the initial user interface (see Figure 19 **(0)**).

B. Implementation using the Rule Generation Concept

The section above just discussed possible reconfigurations and redesigns that could be applied to a user interface for controlling the speed of a water pump in the nuclear reactor simulation. This example shows how the degree of automation could be increased or decreased using the reconfiguration concept introduced in this paper. Still, it has not been addressed how the rule generation process can implement this scenario. Therefore, this subsection discusses the reconfiguration step **(I)** of Figure 19. The needed data will be characterized for instantiating the rule class that will be described in a next step.

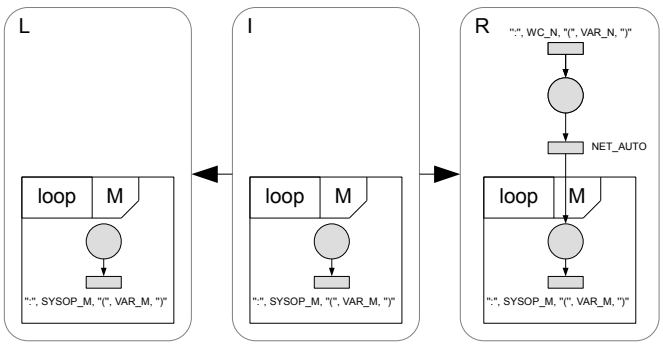

Figure 20. Rule skeleton for deriving the rule necessary for applying reconfiguration step **(I)** as defined in Figure 19.

As presented above, cognitive load is a relevant value to trigger an adaptation of the user interface according to its value. Various works have identified pupillometry as a possible indicator of user's workload, as these by Weert [54], de Greef et al. [55], and Halverson et al. [56]. Thus, the rule generator can be triggered by a component that measures workload through pupillometry. Here, it can be seen that the instantiation of a rule class is accompanied with a use case dependent pre-processing, in this case the mental workload. This is necessary to trigger the rule class instantiation process as defined in Figure 13. In the rule class validation step, the mental workload value is further used to decide, (a) which rule class should be used and (b) whether a selected rule class should be applied to the user interface. The XML snippet given in Appendix E shows one possible implementation of a rule class describing step **(I)**, which would be validated as a rule class that can be applied to the user interface.

In Figure 20, the rule skeleton used in this rule class can be seen, which is the relevant structure, which defines the reconfiguration applied to the user interface in step **(I)**. The class definition shows that the automation model has to be provided from outside as graph (cf. `externReferenceNet` in Appendix E), thus by the tool that triggers the reconfiguration. Furthermore, the system operations (*inc* and *dec* in the above example) as well as the involved interaction elements, which are removed from the physical representation have to be provided by this external tool. Still, the latter could also be provided by a net traversal that identifies the widget transitions connected to the specified system operations. Here, it was decided to reduce the complexity of the rule skeleton by assuming the interaction elements to be provided by the triggering instance. Finally, the subnet `NET_AUTO` has to specify which transition is connected to the newly added interaction element and which transitions are connected to the system operations *inc* and *dec*. This could also be done using the specification of rule skeletons, as has been discussed above. Here, through matching of inscriptions, the correct mapping can be algorithmically derived. The finally derived rule can be seen in Figure 19, which is comprised of the application of the REP and EXT algorithm as discussed above.

VI. Conclusion and Future Work

The paper at hand introduced a new approach to algorithmic rule generation as basis for flexible and formal creation of adaptive user interfaces. The whole approach is based on a formal modeling language called FILL that is algorithmically transformed into reference nets, a special type of Petri nets. This transformation equips FILL with formal semantics as well as making it executable. This formal modeling approach is used to describe interaction logic of a user interface, which is further extended by a proprietary XML-based format describing the physical representation of a user interface. By applying the DPO graph rewriting approach, this kind of user interface model becomes formally adaptable and thereby fulfills the requirement of a self-contained approach for formal modeling and reconfiguration of user interfaces, as has been defined in the beginning.

Nevertheless, the implementation or creation of adaptive user interfaces needs an algorithmic and computer based approach for a flexible creation of adaptation rules applied to a user interface. Therefore, we introduced a new rule generation concept based on an XML specification of rule classes, equipped with a formal description of rule-skeletons based on graph and string grammars. This makes a flexible declaration of rules possible, as has been shown in a concluding example. This example discusses the creation of an adaptive user interface for changing the degree of automation regarding a user interface to control the throughput of a water pump as part of a nuclear power plan simulation. Here, automation becomes a part of the interaction logic of a user interface.

Future work aims at extending the simply structured user interface modeling approach to be more modularized. This makes a separation of dialog and system models in the interaction logic possible. Furthermore, the presented approach will be completely implemented and investigated in an evaluation study. Here, mainly the aspect of adapting a user interface according to measured mental workload will be investigated in cooperation with cognitive psychologists in the context of a working environment. Questions concerning helpful adaptations and restrictions according to changes in the user interface will be the focus of our research. Finally, it is planned to further identify extensions to the rule class and rule skeleton descriptions following from requirements in other usecases rather than in context of adaptive automation.

Appendix

Appendix A - Below, an example for a DPO rule specification is given in its specific XML format, which is used for applying reconfiguration to reference net-based interaction logic. All nets of the rule are specified using PNML, the Petri Net Markup Language. `deleteNet` references the left side of a DPO rule, `interface` denotes the interface graph I of a DPO rule, where `insertNet` dedicates to the right side of a DPO rule. The DPO rewriting approach and the associated rule description concept is subject of discussion in Section III-D.

```
<rule>
  <deleteNet>
    <net>
      <place id="p1"/>
      <place id="p3"/>
      <transition id="t2">
        <inscription>
          <text>guard x==3;</text>
        </inscription>
      </transition>
      <arc id="a1" source="p1" target="t2">
        <inscription>
          <text>x</text>
        </inscription>
      </arc>
      <arc id="a2" source="t2" target="p3">
        <inscription>
          <text>x</text>
        </inscription>
      </arc>
    </net>
  </deleteNet>
  <interface>
    <net>
      <place id="p1"/>
      <transition id="t2">
        <inscription/>
      </transition>
    </net>
  </interface>
  <insertNet>
    <net>
      <place id="p1"/>
      ...
    </net>
  </insertNet>
  <mapping>
    <mapElement insertID="p1"
      interfaceID="p1" deleteID="p1"/>
    ...
  </mapping>
</rule>
```

Appendix B - Below, an example of a rule class specification is given as XML file, where `rc` specifies the namespace for rule classes and `bx` the namespace for boxes in rule skeletons. Rule skeletons are subject of discussion in Section IV-B.

```
<rc:ntdeclaration name="WO"
   rdf:datatype=
     "http://uieditor.org/nttypes/widgetInteract"/>
<rc:ntdeclaration name="WC"
   rdf:datatype=
     "http://uieditor.org/nttypes/newWidget"/>
<rc:ntdeclaration name="Y_N" iterate="N"
   rdf:datatype=
     "http://uieditor.org/nttypes/netTrav"/>
...
<bx:parameter name="N">
  <bx:value
     rdf:datatype=
       "http://uieditor.org/datatypes/int"
       value="2"/>
</bx:parameter>
<bx:parameter name="X">
  <bx:value
     rdf:datatype=
       "http://uieditor.org/datatypes/intInteract"/>
  <bx:description>
    This is a description shown in the interactive
    input box for this value.
  </bx:description>
</bx:parameter>
...
```

Appendix C - Below, an example of the extended PNML format for describing rule skeletons can be seen, where `pnml`

denotes the namespace of PNML and bx the namespace for the box description. Nonterminals are specified as part of inscriptions related to transitions, places, or edges. These do not need an extension of PNML associated to a individual namespace. Rule skeletons are subject of discussion in Section IV-B.

```
<pnml:net>
  <bx:alt>
    <bx:if condition="X>5">
      <pnml:transition id="t1">
        <pnml:inscription>
          <text>":"WC,"(",VAR,")"</text>
        </pnml:inscription>
      </pnml:transition>
      <pnml:arc id="e1"
          source="t1" target"p1">
        <pnml:inscription>
          <text>VAR</text>
        </pnml:inscription>
      </pnml:arc>
      <bx:loop counter="N">
        <pnml:place id="p1"/>
        <pnml:place id="p2"/>
        <pnml:transition id="t2"/>
        <pnml:arc id="e2"
            source="p1" target"t2">
          <pnml:inscription>
            <text>Y_N</text>
          </pnml:inscription>
        </pnml:arc>
        <pnml:arc id="e3"
            source="t2" target"p2">
          <pnml:inscription>
            <text>Y_N</text>
          </pnml:inscription>
        </pnml:arc>
      </bx:loop>
    </bx:if>
    <bx:else>
      ...
      <bx:loop counter="N">
        ...
      </bx:loop>
      <bx:loop counter="N">
        ...
      </bx:loop>
    </bx:else>
  </bx:alt>
</pnml:net>
```

Appendix D - Below, an example of a complete rule class can be seen, which mainly specifies the structure of a rule class description. The introduction and discussion of rule classes can be found in Section IV-C. A concrete example of a rule class is given in Appendix E.

```
<rc:class
    xmlns:bx="http://uieditor.org/boxing/"
    xmlns:rc="http://uieditor.org/ruleClass/"
    xmlns:rule="http://uieditor.org/rule/"
    xmlns:pnml=
      "http://www.pnml.org/version-2009/grammar/"
    name="Interactive Widget Fusion">

  <!-- Class description>
  <rc:description>
    This class specifies the
    interactive fusion of n widgets.
  </rc:description>

  <!-- Selection Heuristic and Net Traversal>
  <rc:select type="first"/>
  <rc:travers type="standard"/>

  <!-- Instantiation precondition-->
  <rc:variable name="..." rdf:datatype="..."/>
  ...
  <rc:precondition con="..."/>
  ...

  <!-- NT and Box Parameter declaration -->
  <rc:ntdeclaration name="..."
      rdf:datatype="..."/>
  ...
  <bx:parameter name="N">
    ...
  </bx:parameter>
  ...

  <!-- Rule skeleton -->
  <rc:ruleSkeleton>
    <rule:deleteNet>
      <pnml:net>
        ...
      </pnml:net>
    </rule:delecteNet>
    <rule:interface>
      ...
    </rule:interface>
    <rule:insertNet>
      ...
    </rule:insertNet>
    <rule:mapping>
      ...
    </rule:mapping>
  </rc:ruleSkeleton>

  <!-- Redesign -->
  <rc:redesign>
    ...
  </rc:redesign>
</rc:class>
```

Appendix E - Below, an example of a rule class can be seen, that is dedicated to a concrete example of adaptive automation. This example is discussed in more detail in Section V-B.

```
<rc:class
    xmlns:bx="http://uieditor.org/boxing/"
    xmlns:rc="http://uieditor.org/ruleClass/"
    xmlns:rule="http://uieditor.org/rule/"
    xmlns:pnml=
      "http://www.pnml.org/version-2009/grammar/"
    name="Automate process">

  <!-- Class description>
  <rc:description>
    Increase automation
  </rc:description>

  <!-- Selection Heuristic and Net Traversal>
  <rc:select type="first"/>
  <rc:travers type="standard"/>

  <!-- Instantiation precondition-->
  <rc:variable name="MW"
    rdf:datatype="mentalWorkload"/>
  <rc:precondition con="MW<2"/>
  <rc:variable name="OP"
    rdf:datatype="int"/>
  <rc:precondition con="OP>=2"/>

  <!-- NT and Box Parameter declaration -->
  <bx:parameter name="M">
    <bx:value
```

```
      rdf:datatype="int"
      value="OP"/>
  </bx:parameter>
  <rc:ntdeclaration name="NET_AUTO"
      rdf:datatype="externReferneceNet"/>
  <rc:ntdeclaration name="SYSOP_M"
      rdf:datatype="sysOpName"/>
  <rc:ntdeclaration name="VAR_M"
      rdf:datatype="sysOp"/>
  <rc:ntdeclaration name="VAR"
      rdf:datatype="widgetInteract"/>
  <rc:ntdeclaration name="WC"
      rdf:datatype="newWidget"/>
  <rc:ntdeclaration name="WO_M"
      rdf:datatype="widgetInteract"/>

  <!-- Rule skeleton -->
  <rc:ruleSkeleton>
    <!-- See Fig. 18-->
  </rc:ruleSkeleton>

  <!-- Redesign -->
  <rc:redesign>
    <rc:newWidget reference="WC"
      widgetType=
        "http://uieditor.org/widgets/button"
      method=
        "http://uieditor.org/widgets/actionEvent"/>
    <rc:deleteWidget reference="WO_M"/>
  </rc:redesign>
</rc:class>
```

REFERENCES

[1] B. Weyers, "User-centric adaptive automation through formal reconfiguration of user interface models," in CENTRIC 2013, The Sixth International Conference on Advances in Human oriented and Personalized Mechanisms, Technologies, and Services, 2013, pp. 104–107.

[2] J. T. Reason and J. T. Reason, Managing the risks of organizational accidents. Ashgate Aldershot, 1997, vol. 6.

[3] A. Jameson, "Adaptive interfaces and agents," in Human-Computer Interaction Handbook. Erlbaum, 2003, pp. 305–330.

[4] K. Reinecke and A. Bernstein, "Improving performance, perceived usability, and aesthetics with culturally adaptive user interfaces," ACM Transactions on Computer-Human Interaction (TOCHI), vol. 18, no. 2, 2011, pp. 1–29.

[5] B. Weyers, D. Burkolter, A. Kluge, and W. Luther, "Formal modeling and reconfiguration of user interfaces for reduction of human error in failure handling of complex systems," International Journal of Human Computer Interaction, vol. 28, no. 10, 2012, pp. 646–665.

[6] B. Shneiderman, "Promoting universal usability with multi-layer interface design," in ACM SIGCAPH Computers and the Physically Handicapped, no. 73-74. ACM, 2003, pp. 1–8.

[7] B. Weyers, Reconfiguration of User Interface Models for Monitoring and Control of Human-Computer Systems. Munich: Dr. Hut, 2012.

[8] P. A. Hancock, R. J. Jagacinski, R. Parasuraman, C. D. Wickens, G. F. Wilson, and D. B. Kaber, "Human-automation interaction research past, present, and future," Ergonomics in Design: The Quarterly of Human Factors Applications, vol. 21, no. 2, 2013, pp. 9–14.

[9] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the gap between modelling and java," in Software Language Engineering. Springer, 2010, pp. 374–383.

[10] S. Decker, S. Melnik, F. Van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks, "The semantic web: The roles of xml and rdf," Internet Computing, IEEE, vol. 4, no. 5, 2000, pp. 63–73.

[11] A. Jameson, "Adaptive interfaces and agents," Human-Computer Interaction: Design Issues, Solutions, and Applications, vol. 105, 2009.

[12] T. Lavie and J. Meyer, "Benefits and costs of adaptive user interfaces," International Journal of Human-Computer Studies, vol. 68, no. 8, 2010, pp. 508–524.

[13] P. Langley and H. Hirsh, "User modeling in adaptive interfaces," Courses and lectures-international centre for mechanical sciences, 1999, pp. 357–370.

[14] G. Fischer, "User modeling in human–computer interaction," User modeling and user-adapted interaction, vol. 11, no. 1-2, 2001, pp. 65–86.

[15] S. Cheng and Y. Liu, "Eye-tracking based adaptive user interface: implicit human-computer interaction for preference indication," Journal on Multimodal User Interfaces, vol. 5, no. 1-2, 2012, pp. 77–84.

[16] G. Kahl, L. Spassova, J. Schöning, S. Gehring, and A. Krüger, "Irl smartcart-a user-adaptive context-aware interface for shopping assistance," in Proceedings of the 16th international conference on Intelligent user interfaces. ACM, 2011, pp. 359–362.

[17] R. Hervás and J. Bravo, "Towards the ubiquitous visualization: Adaptive user-interfaces based on the semantic web," Interacting with Computers, vol. 23, no. 1, 2011, pp. 40–56.

[18] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni, "Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability," ACM Transactions on Computer-Human Interaction (TOCHI), vol. 16, no. 4, 2009, pp. 1–18.

[19] O. Sy, R. Bastide, P. Palanque, D. Le, and D. Navarre, "Petshop: a case tool for the petri net based specification and prototyping of corba systems," Petri Nets 2000, 2000, pp. 77–86.

[20] E. Barboni, C. Martinie, D. Navarre, P. Palanque, and M. Winckler, "Bridging the gap between a behavioural formal description technique and a user interface description language: Enhancing ico with a graphical user interface markup language," Science of Computer Programming, vol. 86, 2013, pp. 3–29.

[21] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "Usixml: A language supporting multi-path development of user interfaces," in Engineering human computer interaction and interactive systems. Springer, 2005, pp. 200–220.

[22] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "Uiml: an appliance-independent xml user interface language," Computer Networks, vol. 31, no. 11, 1999, pp. 1695–1708.

[23] F. de Rosis, S. Pizzutilo, and B. De Carolis, "Formal description and evaluation of user-adapted interfaces," International Journal of Human-Computer Studies, vol. 49, no. 2, 1998, pp. 95–120.

[24] C. Janssen, A. Weisbecker, and J. Ziegler, "Generating user interfaces from data models and dialogue net specifications," in Proceedings of the INTERACT'93 and CHI'93 conference on human factors in computing systems. ACM, 1993, pp. 418–423.

[25] G. Brat, C. Martinie, and P. Palanque, "V&v of lexical, syntactic and semantic properties for interactive systems through model checking of formal description of dialog," in Human-Computer Interaction. Human-Centred Design Approaches, Methods, Tools, and Environments. Springer, 2013, pp. 290–299.

[26] R. Bastide, D. Navarre, and P. Palanque, "A tool-supported design framework for safety critical interactive systems," Interacting with Computers, vol. 15, no. 3, 2003, pp. 309–328.

[27] F. Paternò and C. Santoro, "Integrating model checking and hci tools to help designers verify user interface properties," in Interactive Systems Design, Specification, and Verification. Springer, 2001, pp. 135–150.

[28] D. Navarre, P. Palanque, and S. Basnyat, "A formal approach for user interaction reconfiguration of safety critical interactive systems," in Computer Safety, Reliability, and Security. Springer, 2008, pp. 373–386.

[29] D. Navarre, P. Palanque, J.-F. Ladry, and S. Basnyat, "An architecture and a formal description technique for the design and implementation of reconfigurable user interfaces," in Interactive Systems. Design, Specification, and Verification. Springer, 2008, pp. 208–224.

[30] M. Blumendorf, G. Lehmann, and S. Albayrak, "Bridging models and systems at runtime to build adaptive user interfaces," in Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems. ACM, 2010, pp. 9–18.

[31] J. Criado, C. Vicente Chicote, L. Iribarne, and N. Padilla, "A model-driven approach to graphical user interface runtime adaptation," in Proceedings of the MODELS conference, IEEE. M. Jeusfeld c/o Redaktion Sun SITE, Informatik V, RWTH Aachen., 2010.

[32] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, and R. Valk, "An extensible editor and simulation engine for petri nets: Renew," in Applications and Theory of Petri Nets 2004. Springer, 2004, pp. 484–493.

[33] G. Abowd, R. Beale, A. Dix, and J. Finlay, Human-computer interaction. Prentice Hall, 1996.

[34] L. M. Reeves, J. Lai, J. A. Larson, S. Oviatt, T. Balaji, S. Buisine, P. Collings, P. Cohen, B. Kraal, J.-C. Martin et al., "Guidelines for multimodal user interface design," Communications of the ACM, vol. 47, no. 1, 2004, pp. 57–59.

[35] S. A. White and D. Miers, BPMN Modeling and Reference Guide. Future Strategies Inc., 2008.

[36] O. Kummer, Referenznetze. Logos, 2009.

[37] B. Weyers and W. Luther, "Formal modeling and reconfiguration of user interfaces," in Chilean Computer Science Society (SCCC), 2010 XXIX International Conference of the. IEEE, 2010, pp. 236–245.

[38] A. Schürr and B. Westfechtel, "Graph grammars and graph rewriting systems," RWTH Aachen, Tech. Rep. AIB 92-15, 1992.

[39] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini, "Algebraic approaches to graph transformation. Part II: Single pushout approach and comparison with double pushout approach," in Handbook of graph grammars and computing by graph transformation, G. Rozenberg, Ed. World Scientific Publishing, 1997, ch. 4.

[40] H. Ehrig, K. Hoffmann, and J. Padberg, "Transformation of Petri nets," Electronic Notes in Theoretical Computer Science, vol. 148, no. 1, 2006, pp. 151–172.

[41] H. Ehrig, K. Hoffmann, J. Padberg, C. Ermel, U. Prange, E. Biermann, and T. Modica, "Petri net transformation," in Petri Net, Theory and Applications, V. Kordic, Ed. InTech Education and Publishing, 2008, ch. 1.

[42] J. Stückrath and B. Weyers, "Lattice-extended cpn rewriting for adaptable ui models," in Proceedings of GT-VMT 2014 workshop, in press, 2014.

[43] B. C. Pierce, Basic category theory for computer scientists. MIT press, 1991.

[44] M. Heumüller, S. Joshi, B. König, and J. Stückrath, "Construction of pushout complements in the category of hypergraphs," in Proceedings of the Workshop on Graph Computation Models, ser. GCM '10, Enschede, The Netherlands, 2010.

[45] H. Ehrig and J. Padberg, "Graph grammars and Petri net transformations," in Lectures on Concurrency and Petri Nets, ser. Lecture Notes of Computer Science, J. Desel, W. Reisig, and G. Rozenberg, Eds. Springer, 2004, vol. 3098, pp. 65–86.

[46] M. Weber and E. Kindler, "The petri net markup language," in Petri Net Technology for Communication-Based Systems. Springer, 2003, pp. 124–144.

[47] H. Ehrig, G. Engels, and G. Rozenberg, Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools. world Scientific, 1999, vol. 2.

[48] J. Billington, S. Christensen, K. Van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, "The petri net markup language: Concepts, technology, and tools," in Applications and Theory of Petri Nets 2003. Springer, 2003, pp. 483–505.

[49] R. Parasuraman, T. Sheridan, and C. Wickens, "A model for types and levels of human interaction with automation," IEEE Transactions on Systems, Man, and Cybernetics: Systems and Humans, vol. 30, no. 3, 2000, pp. 286–297.

[50] R. Parasuraman and V. Riley, "Humans and automation: Use, misuse, disuse, abuse," Human Factors, vol. 39, no. 2, 1997, pp. 230–253.

[51] C. Wickens and J. Hollands, Engineering psychology and human performance. Addison Wesley, 1999.

[52] M. Endsley, "Toward a theory of situation awareness in dynamic systems," Human Factors, vol. 37, no. 1, 1995, pp. 32–64.

[53] R. Parasuraman, K. Cosenzo, and E. D. Visser, "Adaptive automation for human supervision of multiple uninhabited vehicles: Effects on change detection, situation awareness, and mental workload," Military Psychology, vol. 21, no. 2, 2009, pp. 270–297.

[54] J. Weert, Ship operator workload assessment tool. Department of mathematics and computer science. Technical University Eindhoven, 2006.

[55] T. de Greef, H. Lafeber, H. van Oostendorp, and J. Lindenberg, "Eye movement as indicators of mental workload to trigger adaptive automation," in Proc. of Augmented Cognition, HCII 2009, 2009, pp. 219–228.

[56] T. Halverson, J. Estepp, J. Christensen, and J. Monnin, "Classifying workload with eye movements in a complex task," in Proceedings of the Human Factors and Ergonomics Society Annual Meeting, vol. 56, no. 1. Sage Publications, 2012, pp. 168–172.