Chapter 10
Formal Description of Adaptable Interactive Systems Based on Reconfigurable User Interface Models

Benjamin Weyers

Abstract This chapter presents an approach for the description and implementation of adaptable user interfaces based on reconfigurable formal user interface models. These models are (partially) defined as reference nets, a special type of Petri nets. The reconfiguration approach is based on category theory, specifically on the double pushout approach, a formalism for the rewriting of graphs. In contrast to the related single pushout approach, the double pushout approach allows the definition of reconfiguration rules that assure deterministic results gained from the rewriting process. The double pushout approach is extended to rewrite colored (inscribed) Petri nets in two steps: first, it has already been extended to basic Petri nets and second, the rewriting of inscriptions has been added to the approach in previous work of the author. By means of a case study, this approach is presented for the interactive reconfiguration of a given user interface model that uses a visual editor. This visual editor is equipped with an XML-based rewriting component implemented in the UIEditor tool, which has been introduced as a creation and execution tool for FILL-based user interface models in Chap. 5. This chapter is concluded with a discussion of limitations and a set of future work aspects, which mainly address the rule generation and its application to broader use cases.

10.1 Introduction

Human users of interactive systems are highly individual. Their needs, skills, and preferences vary, as do task and context. To address the applicability of interactive systems to different tasks, concepts such as task modeling and task-driven development have been investigated in previous work (Paternò 2004, 2012). Among other concepts, adaptive and reconfigurable user interfaces can incorporate the individual needs and preferences of a specific user. Reconfiguration of user interfaces has proved beneficial in various ways, such as increased usability and a decreased num-

B. Weyers (✉)
Visual Computing Institute—Virtual Reality & Immersive Visualization, RWTH Aachen University, Aachen, Germany
e-mail: weyers@vr.rwth-aachen.de

© Springer International Publishing AG 2017

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_10

ber of errors in interacting with the user interface (Jameson 2009). Langley defines an adaptive user interface as "…an interactive software artifact that improves its ability to interact with a user based on partial experience with that user" (Langley and Hirsh 1999 p. 358). Here, Langley refers to the ability of a user interface to adapt to the user based on its interactions with that user by facilitating concepts from machine learning and intelligent systems research. These inferred adaptations must be applied to the user interface, which therefore needs to be reconfigurable. A reconfigurable user interface can be changed in its outward appearance and its functional behavior (physical representation and interaction logic, see Chap. 5).

The formal specifications for reconfiguring user interfaces have been also discussed as formal methods in human-computer interaction. Nevertheless, the proposed solutions lack a fully fledged formalization for the creation, execution, and reconfiguration of user interface models based on one coherent concept (see Sect. 10.2). Such a concept would not only offer the opportunity to formally validate a created or reconfigured user interface model but also enable the documentation and analysis of the reconfiguration process itself. This can be of interest in cases where the user (and not an intelligent algorithm) applies changes to the user interface, especially in cases of safety critical systems (such as that described in Sect. 10.5) but also for other kinds of user interfaces, for example, for intelligent house control or ambient intelligent systems. To use intelligent systems to derive the needed reconfigurations, it is also possible to enable the intelligent algorithm (and not the user) to generate the adaptation rule, thus instigating the formalized change. Therefore, the approach presented here addresses system- as well as user-driven reconfiguration of formal user interface models; the latter will be examined in Sect. 10.5 as this has been the topic of earlier research.

This chapter primarily presents an approach to the description and implementation of reconfigurable user interfaces based on formal, executable, and reconfigurable models of user interfaces. A corresponding modeling method for user interface models is presented in Chap. 5 based on a visual modeling language (called FILL) and including a transformation algorithm to reference nets (Kummer 2009), a special type of colored Petri nets (Jensen and Rozenberg 2012). In addition to the formal semantics provided by this transformation, the transformed model is executable, based on the existing simulator for reference nets called Renew (Kummer et al. 2000). Therefore, the reconfiguration approach presented in this chapter assumes models based on reference nets or colored Petri nets in more general terms. The reconfiguration approach is based on category theory (Pierce 1991), specifically on the double pushout approach (Ehrig et al. 1997), which has been developed for the rule-based rewriting of graphs. It offers the definition of rewriting rules such that they can be applied to graph models in a deterministic form, which is not always the case, as with the single pushout approach (Ehrig et al. 1997). The general approach described here, which is applicable for rewriting any graph, was first extended to Petri nets by Ehrig et al. (1997). Then, Weyers (2012) extended the rewriting to inscribed Petri nets on an algorithmic and implementation basis. Stückrath and Weyers (2014b) extend thisrewriting further by formally specifying the rewriting of inscribed Petri nets,

which extends the double pushout approach through the application of a concept based on lattice theory for the representation of inscriptions.

Furthermore, this chapter discusses the problem of generating rules for rewriting graphs. Rewriting based on the double pushout approach is only a formal and technical basis for the adaptation of user interface models. It offers mainly a formal concept for the modeling of adaptable and adaptive user interfaces such that not only the initial user interface model but also its adaptation is defined formally, which offers formal verification. By storing the applied rules to an initial user interface model, the adaptation process is made completely reproducible and, within certain boundaries, reversible. Nevertheless, rule generation is by design not formally specified. As mentioned above, there are two main options for rule generation: manual generation by the user and generation by an intelligent algorithm. This chapter will present an algorithmic approach, which includes the user in the adaptation process to define which parts of a given user interface model have to be reconfigured and how. Based on this information, an algorithm generates a rule that is applied by the reconfiguration component of the UIEditor (see Chap. 5) to the user interface model. This component is an implementation of the double pushout approach able to rewrite reference nets provided as a serialization in the Petri Net Markup Language (PNML) (Weber and Kindler 2003). For consistency, the reconfiguration concept will refer to the familiar user interface model used in the nuclear power plant case study, which was described in Chap. 4.

The next section presents related work on the formal reconfiguration of user interface models. Section 10.3 offers a detailed introduction of the formal rewriting approach, which will serve as a basis for the description of reconfigurable user interfaces and the core technique for enabling fully fledged formal description of reconfigurable user interface models. Then, Sect. 10.4 presents an interactive approach to rule generation. Rules define the specific reconfiguration applied to a given user interface model. To demonstrate the feasibility of this approach as a combination of formal rewriting and interactive rule generation, Sect. 10.5 presents a case study involving the creation of new functionality and the change of automation offered by an earlier user interface model. The latter was investigated in a previously published user study (Weyers 2012) that found that individualizing a user interface model through formal reconfiguration led to a decrease in user error. Section 10.6 concludes the chapter.

10.2 Related Work

Adaptive user interfaces are an integral part of human-computer interaction research. Various studies have discussed use case-dependent views of adaptive user interfaces, and all have a similar goal: to make interaction between a user and a system less error-prone and more efficient. Jameson (2009) gives a broad overview of various functions of adaptive user interfaces that support this goal. One function he identifies is “supporting system use”. He subdivides this into the functions of “taking over parts of routine tasks”, “adapting the interface”, and “controlling a dialog”, all of which are of interest in the context of this chapter. Lavie and Meyer (2010) iden-

tify three categories of data or knowledge needed for the implementation of adaptive user interfaces: task-related, user-related, and situation-related, that is, related to the situation in which the interaction takes place. They characterize situations as either routine or non-routine. They also discuss level of adaptivity, which specifies how much adaptation can be applied to a given user interface. These functions are provided by various implementations of and studies on adaptive user interfaces. A general overview of task and user modeling is provided by Langley and Hirsh (1999) and by Fischer (2001). However, none of these studies address how the data and knowledge is gathered or described; instead, they concentrate on how it can be used for applying changes to a given formal user interface model in an interactive fashion.

Various examples can be found of the successful implementation of adaptive user interfaces that address the aspects discussed above. For instance, Reinecke and Bernstein (2011) described an adaptive user interface implementation that takes the 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 (2012) discussed an adaptive user interface using eye-tracking data to retrieve users' preferences. Kahl et al. (2011) present a system called SmartCart, which provides a technical solution for supporting customers while shopping. It provides context-dependent information and support, such as a personalized shopping list or a navigation service. Furthermore, in the context of ambient intelligent environments, Hervás and Bravo (2011) present their adaptive user interface approach, which is based on Semantic Web technologies. The so-called ViMos framework generates visualization services for context-dependent information. Especially in the context of ambient assisted living, there are various types of adaptive systems that include the user interface. For instance, Miñón and Abscal (2012) described a framework that adapts user interfaces for assisted living by supporting daily life routines. They focus on home supervision and access to ubiquitous systems.

Previous work has shown that formal models of user interfaces can be adapted to change their outward appearance, behavior, or both without necessarily abandoning the formalization that describes the user interface model. Navarre et al. (2008a, b) described the reconfiguration of formal user interface models based on predefined replacements that are used in certain safety-critical application scenarios, such as airplane cockpits. Blumendorf et al. (2010) introduced an approach based on so-called executable models that changes a user interface during runtime by combining 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 was introduced by Criado et al. (2010).

Thus, adaptive user interfaces play a central role in human-computer interaction and are still the focus of ongoing research. Formal techniques in their development, creation, and reconfiguration are still discussed in the literature, offering various advantages regarding modeling, execution, and verification. Petri net- and XML-based approaches are already in use in various application scenarios. Nevertheless, none of these approaches presents a full-fledged solution for the cre-

ation and reconfiguration of user interface models in one coherent formalization. Furthermore, none of the approaches discusses a closely related concept that enables computer-based systems to generate and apply reconfiguration flexibly and independently of use cases. This chapter introduces a self-contained approach for visual modeling and creation (based on the approach presented in the Chap. 5 on FILL), rule-based reconfiguration, and algorithmic rule generation of user interfaces that builds a formal framework for the creation of adaptive user interfaces.

This approach has previously been explored in various publications. In Burkolter et al. (2014) and Weyers et al. (2012) the interactive reconfiguration of user interfaces models was used to reduce errors in interaction with a simulation of a simplified nuclear reactor. In Weyers et al. (2011), Weyers and Luther (2010), the reconfiguration of a user interface model was used in a collaborative learning scenario. In Weyers (2015), the approach was used to describe adaptive automation as part of a user interface model. Publications specifically focusing on the rule-based adaptation of FILL-based user interface models are Stückrath and Weyers (2014b), Weyers (2012), Weyers et al. (2014), Weyers and Luther (2010).

10.3 Formal Reconfiguration

This section introduces a reconfiguration approach (where reconfiguration refers to the adaptation of interaction logic) that is based on the double pushout approach, which originated with category theory and assumes a formal model of a user interface as has been specified in Chap. 5. Thus, the basic architecture of a user interface model is assumed to differentiate between a physical presentation and an interaction logic. The physical representation comprises a set of interaction elements with which the user directly interacts. Each interaction element is related to the interaction logic, which models the data processing between the physical representation and the system to be controlled. It is further assumed that reconfiguration will be applied to the interaction logic. Nevertheless, in various cases changing the interaction logic also implies changes in the physical representation. This will be the topic of the generation of rules and the case study for the application of reconfiguration described in Sects. 10.4 and 10.5. Before presenting the approach itself, the following description will briefly examine the reasons for using a graph-rewriting approach rather than other means of adapting formal models.

10.3.1 Double Pushout Approach-Based Reconfiguration

As mentioned in the introduction to this section, formal reconfiguration can be differentiated from redesign, where redesign refers to changes in the physical representation of a user interface model, while reconfiguration refers to changes in itsinteraction

logic. Here, it is assumed that the interaction logic is modeled using FILL and then transformed to reference nets. Thus, reconfiguration means changing reference nets, necessitating a method that is (a) able to change reference net models and (b) defined formally to prevent reconfigurations from being nondeterministic. Various graph transformations and rewriting approaches can be found in the literature. Shürr and Westfechtel (1992) identify three different types of graph rewriting systems. The logic-oriented approach uses predicate logic expressions to define rules. This approach is not widespread due to the complexity of its 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 can occur when applying set-theoretical rules to graph-based structures. The third class of techniques is based on graph grammars. Using graph grammars for reconfiguration means changing production rules instead of defining rules to change an existing graph. At a first glance, this seems uncomfortable and counterintuitive for the reconfiguration of interaction logic.

Thus, graph rewriting based on category theory as forth option seems the best starting point for reconfiguration. First of all, pushouts (see Definition 1) as part of category theory are well-behaved when applied to graphs—especially the double pushout (DPO) approach, as discussed by Ehrig et al. (1997). The DPO approach specifies rules that explicitly define which nodes and edges are deleted in the first step and added to the graph in the second. This is not true of the single-pushout (SPO) approach, which is implementation-dependent or generates results that are unlikely to be valid graphs (Ehrig et al. 1997). To give a simple example, the SPO approach can result in dangling edges, which are edges having only a source or a destination but not both. A further problem can be the implicit fusion of nodes, which could have negative implications for the rewriting of interaction logic. These aspects have been resolved in the DPO approach by deleting and adding of nodes and edges explicit and by defining a condition that prevents rules from being valid if they produce dangling edges.

A further argument supporting the use of the DPO approach for rewriting interaction logic is that it has been extended and discussed in the context of Petri nets as introduced by Ehrig et al. (2006, 2008), who offer a solid basis for the reconfiguration of reference net-based interaction logic. Another argument for choosing the Petri net-based DPO approach as described by Ehrig et al. is that it can be easily extended to colored Petri nets. Within certain boundaries, the semantics of the inscription can also be taken into account, as described in detail by Stückrath and Weyers (2014b). Here, the treelike structure of the XML-based definition of inscription is ambiguous, which is discussed in greater detail in Sect. 10.3.2.

Like the SPO, the DPO relies on the category theory-based concept of pushouts. Assuming a fundamental understanding of category theory (otherwise consider, e.g., Pierce 1991), a pushout is defined as follows.

Definition 1 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 the *pushout object* of (f, g), and it is true that

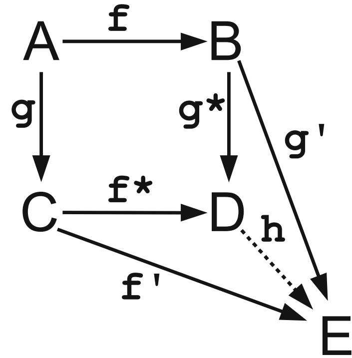

Fig. 10.1 A pushout diagram

1. $g^* \circ f = f^* \circ g$, and
2. for all other objects E with the arrows $f' : C \rightarrow E$ and $g' : B \rightarrow E$ that fulfill the former constraint, there has to be an arrow $h : D \rightarrow E$ 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, whether via B or via C. The second condition guarantees that D is unique, except for isomorphism. Thus, defining (f, g) there is exactly one pushout (f^*, g^*, D) where D is the rewritten result, also called the pushout object. In general, A and B are produced by 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. The resulting diagram is shown in Fig. 10.1.

To define the DPO approach, the *pushout complement* has to be defined first.

Definition 2 Given two arrows $f : A \rightarrow B$ and $g^* : B \rightarrow D$, the triple $(C, g : A \rightarrow C, f^* : C \rightarrow D)$ 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 3 A *matching* is a mapping $m : L \rightarrow G$; a *production* is a mapping $p : L \rightarrow R$, where L, R, and G are graphs. The corresponding mappings of m and p are defined as mapping $m^* : R \rightarrow H$ and $p^* : G \rightarrow H$, where H is also a graph.

Definition 4 A *DPO rule* s is a tuple $s = (m, (l, r), L, I, R)$ for the transformation of a graph G, with $l : I \rightarrow L$ and $r : I \rightarrow R$, which are two total homomorphisms representing the production of s; $m : L \rightarrow G$ 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, the pushout complement C can first be calculated using L, I, m, and l with a given graph G to be rewritten. In the DPO approach, this step deletes nodes

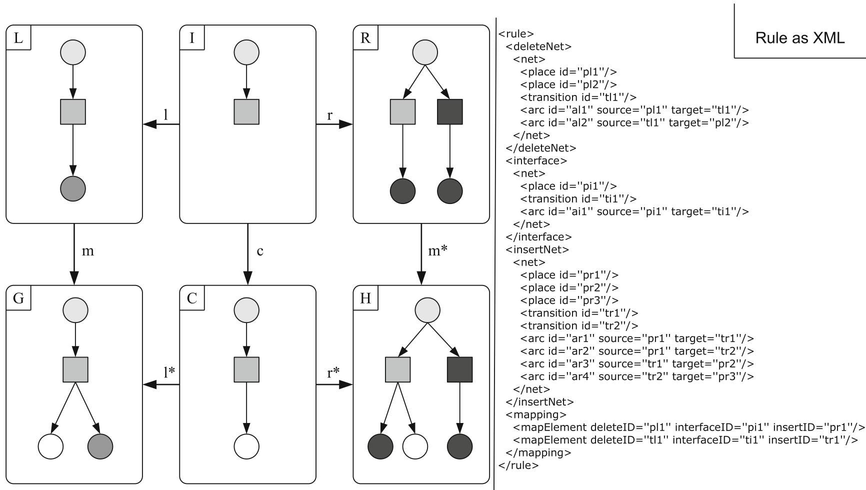

Fig. 10.2 Example for a DPO rule showing its application to a Petri net G

and edges from G. Second, 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. Finally, 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 Fig. 10.2.

Nevertheless, the pushout complement is not unique in all cases and probably does not even exist. However, if the total homomorphisms l and m fulfill the *gluing condition* given below, the pushout complement can be considered to exist. The gluing condition is defined as follows.

Definition 5 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:

$$\nexists e \in (E_G \setminus m(E_L)) : s_G(e) \in m(V_L \setminus l(V_I)) \vee t_G(e) \in m(V_L \setminus l(V_I)), \quad (10.1)$$

and

$$\nexists x, y \in (V_L \cup E_L) : x \neq y \wedge m(x) = m(y) \wedge x \notin l(V_I \cup E_I). \quad (10.2)$$

Condition 10.1 is called *dangling condition*. The homomorphism l of a DPO rule that defines which nodes are 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 10.2 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 in 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 applying 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. 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 in the case of isomorphy. This is further discussed by Heumüller et al. (2010) and in Weyers (2012, p. 107).

Finally, rule descriptions have to be serialized in a computer-readable form. This is necessary for the implementation-side use of rewriting rules presented in the next section. Therefore, an existing XML-based description language for Petri nets (PNML Weber and Kindler 2003) has been used with an extension that structures and describes the rule-specific elements, such as indicating L as `deleteNet`, I as `interface`, and R as `insertNet`. Thus, every net (as embedded in a $< net >$ node) is given as a PNML-based description of the respective left, interface, or right graph of the DPO rule. The $< mapping >$ node specifies l and r as a set of $< mappingElements >$ that are representations of tuples of XML ids. Mapping to G is not part of the rule serialization because a rule is first and foremost independent from a graph being rewritten. An example of such an XML-based representation of a rule can be seen in Fig. 10.2 on the side of the DPO diagram, which also shows the rule applied to a Petri net G.

10.3.2 Rewriting Inscriptions

For the rewriting of inscriptoins, the rewriting approach introduced previously involves only two steps. First, the node that carrying the inscription to be rewritten is deleted. Second, a new node carrying the new inscription is added. This new node must have the same connection as the deleted node; otherwise, the rewriting would also change the structure of the graph, which should not be the case since only the inscription is being rewritten. The problem with this approach is that the rule has to map all incoming and outgoing edges, which can increase the effort involved in generating rules. If this is not done carefully, edges will be delete that are not mapped to prevent dangling edges, generating unintended changes in the net. Furthermore, detailed rewriting of inscriptions offers finer-grained changes to be applied to a Petri net. Thus, slight changes can be made in, for example, guard conditions without the need to rewrite the structure of the net.

Therefore, the previous rewriting approach has been extended such that the rule is not only aware of the nodes and edges in the graph but also of the node's inscriptions. The complete definition and proof of this extension can be found in Stückrath and Weyers (2014a, b). The discussion in this section will focus on deletion-focused PNML-based rewriting. It is assumed that the net and the inscriptions are given as

PNML-based serialization, as introduced above. An XML-based inscription can be formally defined as follows.

Definition 6 Let $(Val, \preceq)$ be a disjoint union of complete lattices Val_i of values with $\biguplus_{i \in I} Val_i = Val$ and let N be a set of IDs sorted such that it can be partitioned in N_i with $N = \biguplus_{i \in I} N_i$. An XML inscription $xml_{N,Val}$ is a directed rooted tree (V, E, r, γ) of finite height, where V is a set of vertices (nodes), $E \subseteq V \times V$ is a set of edges, $r \in V$ is the root and $\gamma : V \rightarrow \bigcup_{i \in I}(N_i \times Val_i)$ maps properties to each vertex. Additionally, for every two edges $(v_1, v_2), (v_1, v_3) \in E$ with $\gamma(v_i) = (n_i, w_i)$ (for $i \in \{2, 3\}$), it holds that $n_2 \neq n_3$.

For every $v \in V$ we define $v\downarrow = (V', E', v, \gamma')$ to be the subtree of $xml_{N,Val}$ with root v, which is an XML inscription itself.

Thus, an XML-based inscription is nothing but a tree structure of nodes and edges, which can be organized into subtrees in a well-defined manner. For the rewriting, it is important to define an order for these trees, so that it is clear whether a subtree is part of another subtree and whether this second subtree is smaller or larger. This is important for deciding what is to be deleted or added and thus for calculating the differences between right side, interface, and left side of the rule. The order is defined on a lattice basis.

Definition 7 Let $XML_{N,Val}$ be the set of all XML inscriptions $xml_{N,Val}$. We define the ordered set $(XML_{N,Val}, \sqsubseteq)$, where for two elements $(V_1, E_1, r_1, \gamma_1) \sqsubseteq (V_2, E_2, r_2, \gamma_2)$ holds if and only if $\gamma_i(r_i) = (n_i, w_i)$ for $i \in \{1, 2\}$. Then $n_1 = n_2$, $w_1 \preceq w_2$, and for all $v_1 \in V_1$ with $(r_1, v_1) \in E_1$, there is a $v_2 \in V_2$ with $(r_2, v_2) \in E_2$ such that $v_1\downarrow \sqsubseteq v_2\downarrow$.

10.3.2.1 Deletion-Focused Rewriting

Deletion-focused rewriting refers to rewriting that prefers the deletion of an inscription and thereby prevents undefined behavior by the rewriting rule. An example can be seen in Fig. 10.3. The rewriting is based on the DPO approach, calculating deletion and adding elements in the inscription by first computing the pushout complement δ' and then computing the pushout object δ''. What is to be deleted or added (as with the DPO net rewriting) is identified based on the definition of the difference between the left side and the interface graph of the rule (here α and β) and between the interface graph and the right side (here β and γ). As defined above, these differences are derived using the lattice-based specification of (XML) tree-structured inscriptions. Furthermore, the pushout complement and the pushout itself are calculated using a strategy called *deletion-focused rewriting*. This strategy is needed to prevent the rewriting of undefined behavior if the pushout complement is not unique (see discussion in the previous section). In deletion-focused rewriting, the pushout complement with the smallest result (defined by the lattice) is chosen.

To make this strategy clearer, Fig. 10.3 shows an example of the deletion-focused rewriting of a net using the rule and rewriting shown in the upper part of the figure. The mapping of the left side to the net to be rewritten is shown as hatching pattern,

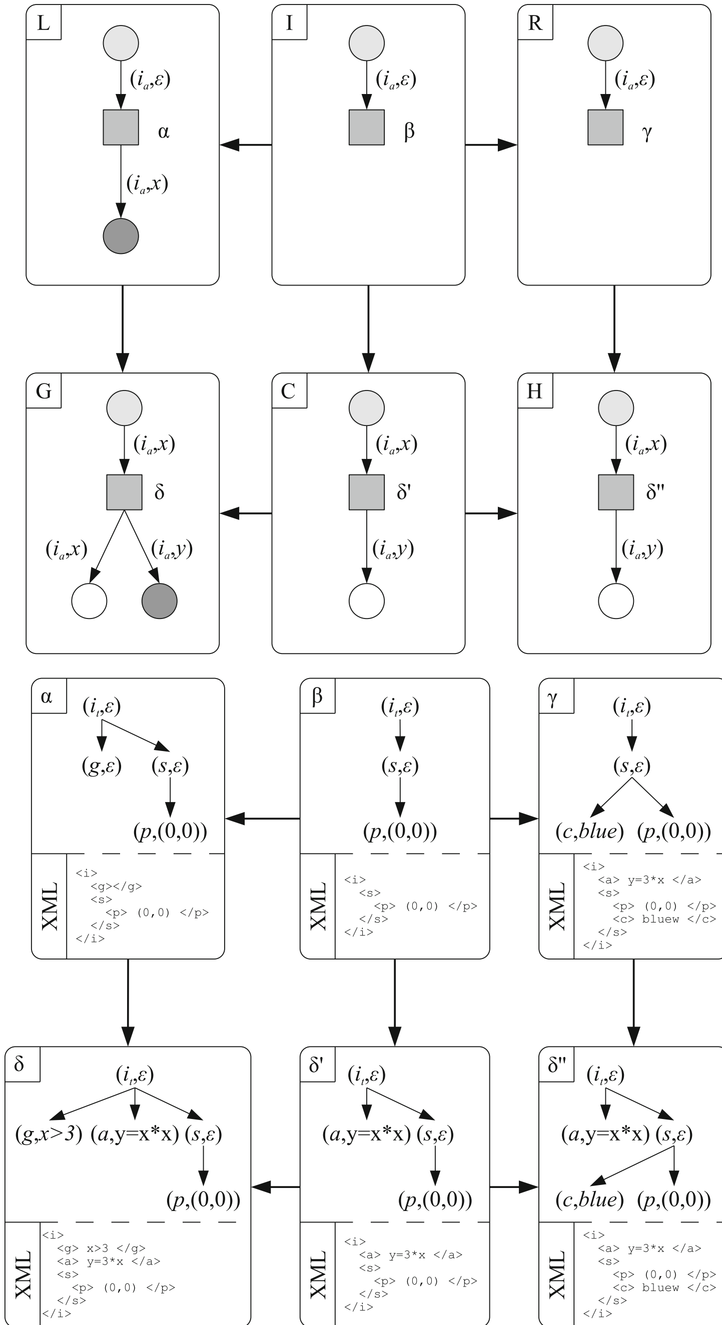

Fig. 10.3 Example for rewriting of an XML-based inscription of a petri net. The *upper half* shows the DPO rule for rewriting the net; the *lower half* shows the DPO rule based on lattice-structured inscription for rewriting the inscription of one transition (*middle gray*) in the upper DPO rule (see α, β, γ and δ to δ'')

where a similar pattern defines the mapping of two nodes (in L and G) to one another. in the lower part of Fig. 10.3, the rewriting of the inscription of a transition (vertical hatching) is visible. Various types of XML nodes are involved, such as the inscription node $< i >$, guard condition $< g >$, style node $< s >$ and nodes for color $< c >$ and position $< p >$. The rule assumes that, after the deletion of the node (g, e) (specified as difference between α and β), the resulting inscription node no longer has a guard condition, and it therefore prefers to delete rather than preserve an inscription. One could assume a preservation-focused rewriting, which deletes only nodes that are equal to or smaller than the matched one. In the present case, the matched node is larger and thus contains non-empty content ($x > 3$); therefore, it would not be deleted because the preservation-focused rewriting rule would be inapplicable (Stückrath and Weyers 2014b).

With deletion-focused rewriting, the outcome is different. Nodes are also deleted if they are larger as shown in the example. The rule specifies the deletion of the guard inscription node such that the push-out complement δ', as shown in Fig. 10.3, is derived. In this case, the complete node will be deleted. In addition to the deletion of the guard condition $(g, x > 3)$ from graph δ, the rule specifies adding a color to the style inscription node. Similar to the rewriting of Petri nets, the rules specify a new subnode of the (s, ϵ) node. Applying this adding operation to δ' yields the graph δ''. The result is a style node defining a position $(0, 0)$ and a color *blue* for the inscribed transition.

10.4 Interactive Reconfiguration and Rule Generation

The previous sections introduced the basic theory of rule-based rewriting of colored Petri nets based on the DPO approach including an extension for deletion-focused rewriting of tree-structured inscriptions. This section will introduce an approach for generating rules in a given context including the use of explicit user input and using algorithms to generate rules for rewriting the interaction logic based on user input. It will also explore how rewriting interaction logic influences the physical representation of a user interface model.

The interactive reconfiguration of formally specified user interfaces based on FILL is a component of the UIEditor (see Chap. 5). This component implements algorithms for the generation of rules as well as a visual editor that enables the user to apply these reconfiguration operations to a given user interface model. With this editor, the user can select individual or a group of interaction elements and apply a reconfiguration operation to this selection. Thus, this interactive approach to rule generation uses the physical representation as a front end for the user to define which parts of the underlying interaction logic are to be reconfigured.

The user interface of this visual editor is shown in Fig. 10.4. The workspace presents the physical representation of the user interface to be reconfigured. The tool bar at the top right of the editor window offers a set of reconfiguration operations. For instance, the user can select two buttons and choose a reconfiguration operation

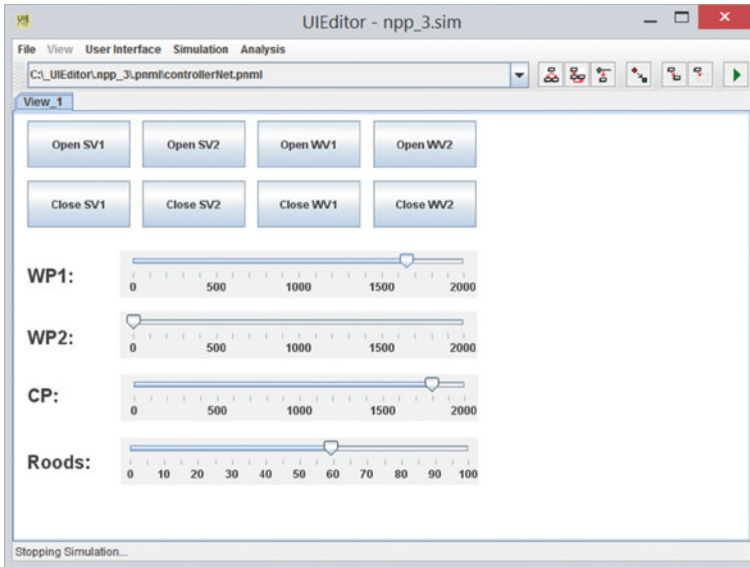

Fig. 10.4 Reconfiguration editor of the UIEditor showing an exemple user interface. In the *upper right corner* is a tool bar with various buttons for the applying reconfiguration operations to a selection of interaction elements

that creates a new button that triggers the operations associated with the two selected buttons in parallel with only one click. The selected buttons indicate which part(s) of the interaction logic must be rewritten. The specific parts of the interaction logic affected are determined by simple graph-traversing algorithms. Selecting the operation (here, `parallelization`) specifies the kind of extension or change to be applied to the interaction logic.

An rule resulting from the aforementioned interactive rule generation can be seen in Fig. 10.5 on the left. Here, the user selected the buttons labeled `Input A` and `Input B`. By applying the parallelization operation to this selection, the underlying algorithm generated the rule shown in the lower half of Fig. 10.5. Applying this rule to the interaction logic creates the net on the right labeled "Reconfiguration", which is also related to a new interaction element, here a newly added button. This example shows that this kind of reconfiguration always implies a change in the physical representation—here, the addition of a new button that triggers the newly created part of the interaction logic.

Another example of an implemented reconfiguration operation is shown in the middle of Fig. 10.5, where the `discretization` of a continuous interaction element is shown. The term *continuous* refers to the types of values that are generated via an interaction element—here, a slider is used to generate values in a certain range. In contrast to a slider, which can generate any of a range of values, a button is a discrete interaction element and can only generate one specific value: an event. Thus, `discretization` is a reconfiguration operation that maps a continuous

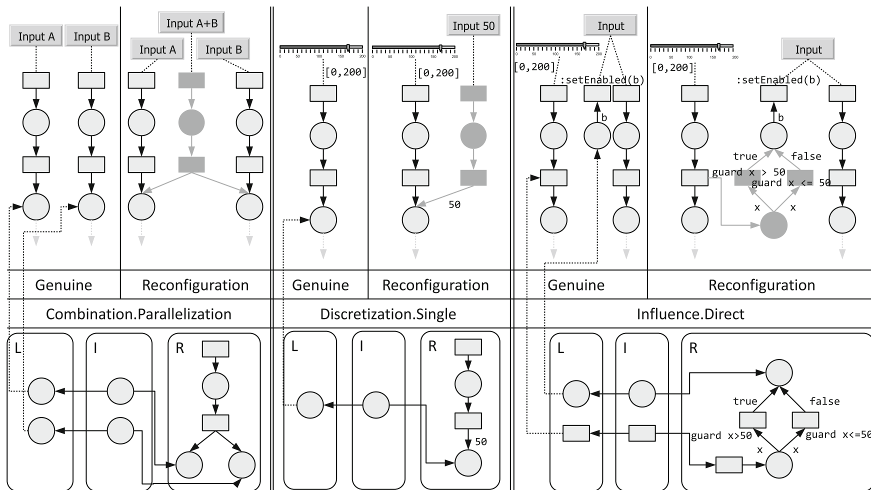

Fig. 10.5 Three examples of reconfiguration operations applied to a user interface. *Left* application of a `parallelization` operation to two buttons, creating a new button that combines the functions of the original two; *middle* application of a `discretization` operation to a slider generating a button that sets a preset value to the system parameter controlled by the slider; *right* application of a `direct` reconfiguration operation that influences the behavior of two interaction elements by making them dependent on each other

interaction element to a discrete one. In Fig. 10.5, a slider is mapped to a button. The button generates a single value from the slider's interval—here, the value 50.

The last example presented in Fig. 10.5 is of a type of rule that cannot be generated interactively. Here, a certain part of the underlying interaction logic of an interaction element is changed. The added subnet changes the status of the button `Input`. Based on the value selected with the slider, the button is disabled or enabled: If the value is ≤ 50, it will be disabled; if the value is >50, it will be enabled. This type of reconfiguration operation can be pre-defined or determined by using extended methods for rule generation, as discussed in Weyers et al. (2014) in detail.

10.5 Case Study

To demonstrate how interaction logic can be rewritten, this section discusses its use in the nuclear power plant case study (see Chap. 4). This user interface model (as partially described in Chap. 5) includes a simple interaction logic that triggers the operations as indicated by the interaction elements' labels (see Fig. 10.6). However, the user interface used in the study offers no complex operations such as the SCRAM operation for emergency shutdown the reactor. Such an operation can be added to the user interface model as an extension of its interaction logic; this process will be

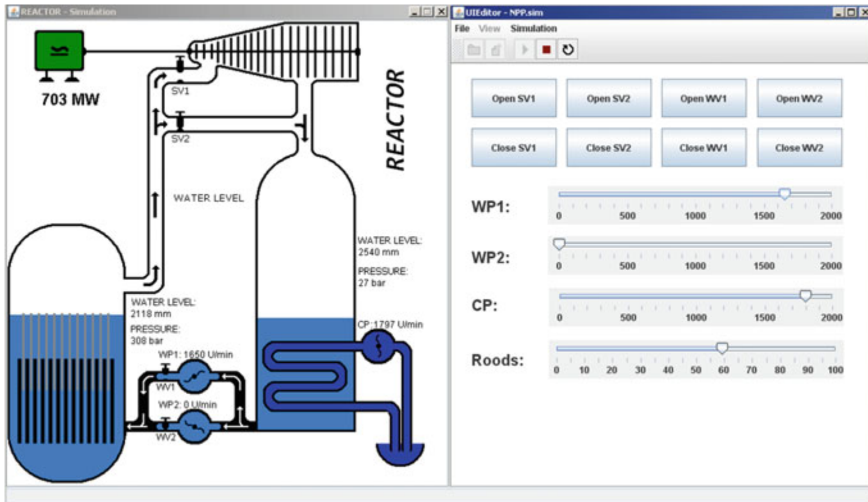

Fig. 10.6 The initial user interface for the control of a simplified simulation of a nuclear power plant running in UIEditor

explained in the next subsection. The following subsection examines the effect of a user-driven reconfiguration on the basis of a user study. A final subsection discusses these results and various aspects of future work.

10.5.1 Case Study: SCRAM Operation

The so-called SCRAM operation is currently missing as part of the user interface. As described in Chap. 4, the SCRAM operation shuts down the reactor immediately and returns it to a safe state. This includes the complete insertion of the control rods into the reactor, which controls the thermal output of the core and stops the chain reaction. The pumps have to continue to cool the core, and the turbine should be shut off from the steam circuit. Finally, the condenser pump has to continue working to condense the steam still being produced by the reactor and thus cool the core.

To implement the SCRAM operation into the given user interface model, users could apply a limited number of reconfigurations to the user interface. Firstly, they could select a slider to define the position of the control rods and apply the `discretization` operation. Here, selecting `0` created a button that set the position of the control rods to 0, such that they were completely inserted in the core. Thus, by pressing the newly created button, the chain reaction would be stopped and the core would produce only residual heat.

Second, the Buttons to open SV2, close SV1, open WV1 and close WV2 should be combined to one button by applying the `parallelization` operation. Thus, the

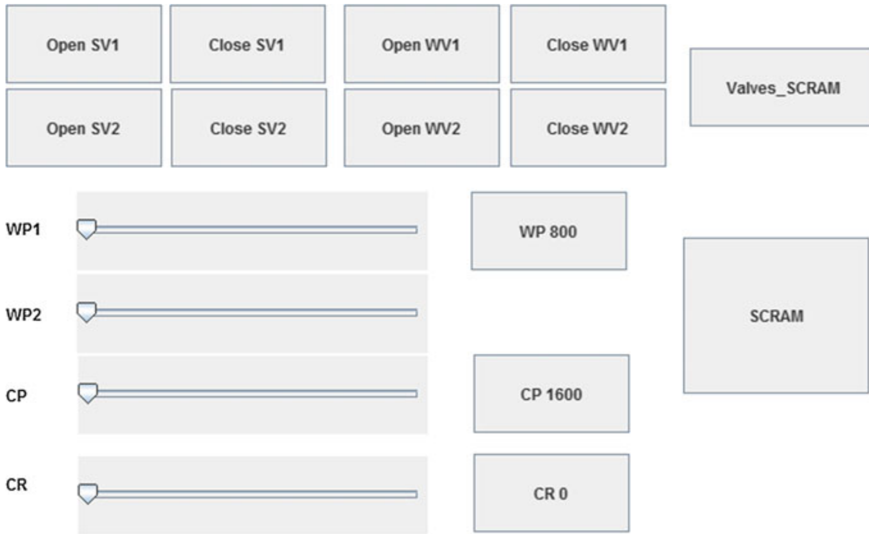

Fig. 10.7 A user interface showing the various operations needed for a SCRAM operation, which has been also added. The user interface shows the result of the application of various reconfiguration operations to an initial user interface through interactive and sequential application of reconfiguration operations

new Button subsumes all relevant valve operations needed for the SCRAM operation. Third, the water pumps WP1 and CP have to be set to a certain speed: WP1 to 800 rpm and CP to 1600 rpm. Therefore, the user selects first the slider for WP1 and applies the `discretization` operation as she did for the control rod position. She repeats this with the slider for CP. The final step is to fuse all the newly created Buttons into one using the `parallelization` operation. The resulting physical representation of the created user interface can be seen in Fig. 10.7.

10.5.2 User Study: Error Reduction Through Individualization

In addition to the extension of the user interface's functionality by creating new interaction elements extending the interaction logic, the effect of such extension on the interaction process between the human user and the system must also be analyzed. To do this, a user study was conducted to measure the effect of the individualization of a given user interface model on the number of errors users committed while working with that interface. For the user study, participants had to control the simplified simulation of the nuclear power plant. The nuclear power plant simulation had been selected to keep the attention of the participants as high as possible (because they

all know about the criticality of faulty control of nuclear plants) and to be as flexible as possible regarding the degree of complexity for the control task. The control process was simple to adapt for the study in that it could be learned in a limited time but was complex enough to make any effects visible. The results of the study indicated a reduction of errors in controlling the power plant when participants used their individually reconfigured user interfaces (Burkolter et al. 2014; Weyers et al. 2012).

In total 72 students of computer science (38 in the test group) participated in the study. First, all participants responded to a pre-knowledge questionnaire to ensure that the groups were balanced regarding knowledge about nuclear power plants. Next, all participants were introduced to the function of a nuclear power plant and its control. This introduction was a slide-based presentation that was read aloud to keep content and time equal among all groups of participants. Afterwards, participants participated in an initial training session on the controls for the nuclear power plant simulation; the instructions were presented on paper. In total three procedures were practiced: starting the reactor, shutting it down, and returning the reactor to a safe state after one of the feedwater pumps failed. Participants were allowed enough time that every participant was able to complete each procedure at least once. After the training in each procedure, the participants in the test group had the chance to apply reconfiguration operations to the initial state of the user interface and test their changes. For this, they had a specific amount of time after each training run. To keep time on task equal for both the test and the control group, the control group watched a video on simulation in general (not of nuclear plants) for the same amount of time as the test group had to apply reconfigurations to their user interface. To apply changes to the initial user interface, they used the interactive rule generation procedure as implemented in UIEditor. An example of an outcome of this kind of individualization can be seen in Fig. 10.8.

After the training and reconfiguration phases, the participants had to start up and shut down the reactor several times, simulating real use of the system. In the last runthrough, a feedwater pump broke down. Participants had been trained in the response to this event, but had not been informed in advance that it would occur.

During the trials, errors were measured in the sense of wrongly applied operations. An operation was classified as wrongly applied if it did not match the expected operation for the startup or shutdown of the reactor or for the response to the water pump breakdown in the final runthrough. Various types of errors were detected, based on the classification defined by Hollnagel (1998). All error types identified are shown in Fig. 10.9, which also shows how the log files were evaluated by manually looking for patterns and mismatches.

The results show that the control group using the individualized user interface models made fewer errors in interacting with the reactor than did the control group, which used the initial user interface without individualization. Furthermore, the test group was able to respond to the system failure more effectively than the control group. These results are shown in Table 10.1. This study shows the potential for this type of adaption of user interface models. It has been previously published in Burkolter et al. (2014) and Weyers et al. (2012).

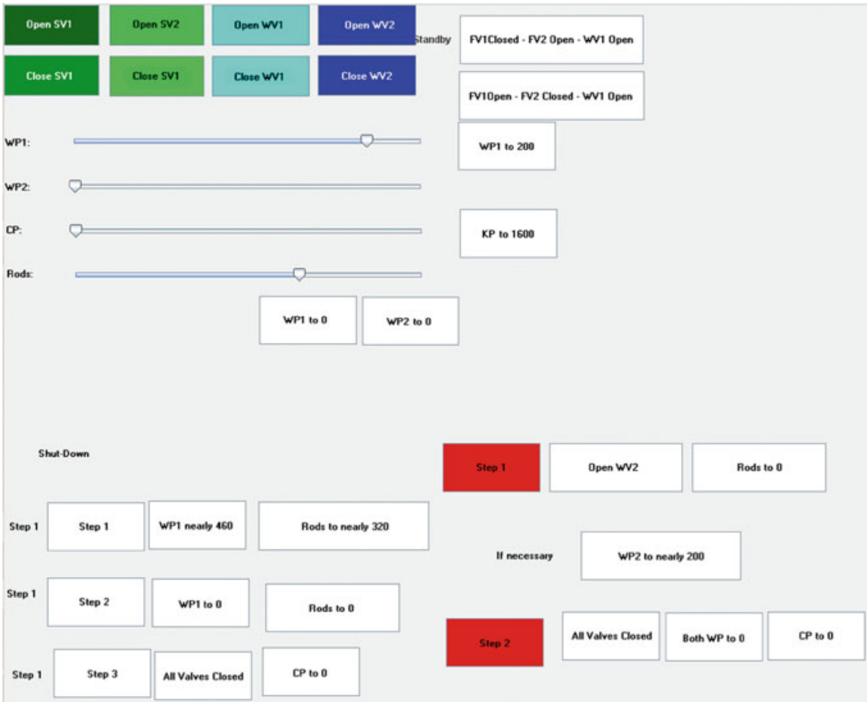

Fig. 10.8 An example of a reconfigured user interface for the control of a simplified simulation of a nuclear power plant

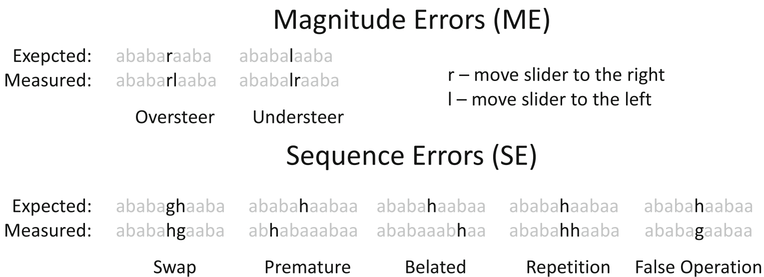

Fig. 10.9 Different error types evaluated. The errors were identified by comparing the interaction logs gathered during the use of the system with the expected operation sequences, which were provided to the participants as control operation sequences on paper

10.5.3 Discussion

In addition to the benefits of a user-driven reconfiguration of a user interface model as shown above, there are various side conditions to be considered. First, by adding more abstract operations to the user interface, the degree of automation is increased

Table 10.1 Startup of reactor and practiced system fault: mean, standard deviation (in parentheses), and results of the applied t-test for comparing the means of the group using the initial user interface (NonRec-group) and the grouping using their individualized/reconfigured user interface (Rec-group)

Error types	Rec-group	NonRec-group	t(df), p (One-tailed), ES r
Magnitude error			
Oversteering	0.38 (0.70)	2.38 (1.72)	t(45.86) = −6.13, **p = 0.000**, r = 0.67
Understeering	0.15 (0.37)	0.15 (0.36)	t(58) = 0.07, p = 0.472, r = 0.01
Sequence error			
Swap	0.04 (0.20)	0.47 (0.86)	t(37.40) = −2.83, **p = 0.004**, r = 0.42
Premature	0.23 (0.43)	0.24 (0.50)	t(58) = −0.04, p = 0.486, r = 0.01
Belated	0.42 (0.58)	0.38 (0.70)	t(58) = 0.24, p = 0.405, r = 0.03
Repetition	0.38 (0.70)	0.09 (0.29)	t(31.54) = 2.04, **p = 0.003**, r = 0.34
False operation	0.96 (1.25)	1.82 (1.90)	t(56.87) = −2.12, **p = 0.020**, r = 0.27
Total	2.48 (1.93)	5.53 (2.71)	t(59) = −4.93, **p = 0.000**, r = 0.54

which is known to have a potentially negative influence on task performance. This is especially true in critical situations (Parasuraman et al. 2000). However, it is questionable at which point this also becomes true for automation added to the user interface model by the users themselves. The study described argues for a higher task performance through user-driven increases in automation. However, there is no data on whether this is still true if the user works with the reconfigured user interface over a longer period of time. The interactive process of creating more abstract operations could also have a certain effect on training and user understanding of the interactive system controls. The effects of increased automation and the enhancement of training through interactive reconfiguration are aspects to be considered in future work.

A related problem is the creation of erroneous operations which a user could implement while individualizing the user interface. Erroneous operations could result in unexpected system operation or faulty system states. The scenario discussed above does not prevent users from adding erroneous operations to the user interface model. However, the underlying seamless formalization of the user interface model and the reconfiguration process constitute the basis for a formal validation of the assumed outcome of a reconfiguration. Thus, the user can be informed of potential problems arising from the application of a certain reconfiguration rule. Such a validation approach is also of interest for automatic non-user-driven reconfiguration. Nevertheless, the development of "on-the-fly" validation is not trivial, especially if the underlying system is not known. Initial studies on this subject have been conducted by facilitating a SMT solver (Nieuwenhuis et al. 2006) for the validation of interaction logic. Here, the generated reference net was transformed algorithmically into an expression of first-order logic. This expression was composed by the pre- and post-conditions for the transitions in the reference net and is being tested by the SMT

solver for satisfactory performance. It could be extended by certain pre-conditions relevant to the use case, making it possible to check whether an extended interaction logic generates input values to the system that are not valid or specified for the system to be controlled.

The approach shown offers other benefits related to creation rather than the use of such user interface models. By formalizing the user interface model and its reconfiguration, both models and reconfiguration rules can be reused and facilitated as provenance information describing the reconfiguration process. If the underlying system is replaced by, for example, a newer version or a similar system, the user interface model can be reused as a whole—assuming, of course, that there are no changes in the system interface providing the access to the system values to be controlled. However, if changes have been applied to that interface, these can simply be adopted in the user interface model as well without the need to touch any code. It is imaginable that such changes could be applied automatically by applying reconfiguration rules gathered from a (hopefully formalized) description of the API changes.

The user study presented addresses only the use of reconfigured user interfaces by using the interactive rule generation approach discussed in Sect. 10.4. No deeper evaluation of UIEditor has been conducted, especially addressing the creation of user interface models. The exception is results gathered from items in the post-study questionnaire of the study described above, which indicated no problems with the reconfiguration concept. However, an important goal of future work is to intensify the research and development effort on measuring and enhancing the usability and user experience of UIEditor. The use of UIEditor by non-experts in HCI and user interface modeling is also of great interest. This user group, which would include mechanical engineers among others, would benefit from the simplified creation and rapid prototyping of user interfaces without having to resign the benefits of formal validation and reuse of such models.

As these case studies show, the reconfiguration approach is well-suited for the individualization of user interface models. Nevertheless, the approach offers the algorithmic generation of reconfiguration rules by, for example, intelligent algorithms, as is the case with adaptive or intelligent interactive systems (Langley and Hirsh 1999). An initial approach to the automatic generation of reconfiguration rules was presented in Weyers et al. (2014). Such an extension to the current approach could also enable its integration into frameworks like CAMELEON (Calvary et al. 2003; Pleuss et al. 2013). The CAMELEON framework focuses mainly on the application of abstract user interface models to specific contexts and systems in final user interface models. Nevertheless, it does not address the underlying functionality, that is, the interaction logic of a user interface. The approach presented here could embed interaction logic into the transformation from abstract to concrete user interface models. Starting with an abstract interaction logic and reconfiguring it step-wise using rule-based rewriting and applying the transformation to the physical representation makes that interaction logic specific to a certain context, user, task, or device.

10.6 Conclusion

This chapter presents a solution for the reconfiguration of formal user interface models based on reference nets, a special type of Petri nets. This reconfiguration is based on a graph rewriting approach called the double pushout approach, which has been extended to inscribed/colored Petri nets. This extension was based on lattice theory to define a formal structure for XML node and edge inscription, making it possible to rewrite inscriptions in such a way that it is not necessary to apply any changes to the net structure. The applicability of this approach was demonstrated in a case study in which new operations were added to an initial user interface for the control of a simplified nuclear power plant simulation. For this reconfiguration, an interactive rule generation approach was facilitated which enabled the user to specify the relevant parts of the user interface to be reconfigured as well as the type of reconfiguration to be applied. This tool is part of UIEditor as introduced in Chap. 5.

References

Blumendorf M, Lehmann G, Albayrak S (2010) Bridging models and systems at runtime to build adaptive user interfaces. In: Proceedings of the 2nd ACM SIGCHI symposium on engineering interactive computing systems, Berlin

Burkolter D, Weyers B, Kluge A, Luther W (2014) Customization of user interfaces to reduce errors and enhance user acceptance. Appl Ergon 45(2):346–353

Calvary G, Coutaz J, Thevenin D, Limbourg Q, Bouillon L, Vanderdonckt J (2003) A unifying reference framework for multi-target user interfaces. Interact Comput 15(3):289–308

Cheng S, Liu Y (2012) Eye-tracking based adaptive user interface: implicit human-computer interaction for preference indication. J Multimodal User Interface 5(1–2):77–84

Criado J, Vicente Chicote C, Iribarne L, Padilla N (2010) A model-driven approach to graphical user interface runtime adaptation. In: Proceedings of the MODELS conference, Oslo

Ehrig H, Heckel R, Korff M, Löwe M, Ribeiro L, Wagner A, Corradini A (1997) Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach. In: Rozenberg G (ed) Handbook of graph grammars and computing by graph transformation. World Scientific Publishing, Singapore

Ehrig H, Hoffmann K, Padberg J (2006) Transformation of Petri nets. Electron Notes Theor Comput Sci 148(1):151–172

Ehrig H, Hoffmann K, Padberg J, Ermel C, Prange U, Biermann E, Modica T (2008) Petri net transformation. In: Kordic V (ed) Petri net, theory and applications. InTech Education and Publishing, Rijeka

Fischer G (2001) User modeling in human-computer interaction. User Model User-Adap Interact 11(1–2):65–86

Hervás R, Bravo J (2011) Towards the ubiquitous visualization: adaptive user-interfaces based on the semantic web. Interact Comput 23(1):40–56

Heumüller M, Joshi S, König B, Stückrath J (2010) Construction of pushout complements in the category of hypergraphs. In: Proceedings of the workshop on graph computation models, Enschede

Hollnagel E (1998) Cognitive reliability and error analysis method (CREAM). Elsevier, Amsterdam

Jameson A (2009) Adaptive interfaces and agents. Hum Comput Interact Des Issues Solut Appl 105:105–130

Jensen K, Rozenberg G (2012) High-level petri nets: theory and application. Springer, Berlin

Kahl G, Spassova L, Schöning J, Gehring S, Krüger A (2011) Irl smartcart-a user-adaptive context-aware interface for shopping assistance. In: Proceedings of the 16th international conference on intelligent user interfaces, Palo Alto

Kummer O (2009) Referenznetze. Logos, Berlin

Kummer O, Wienberg F, Duvigneau M, Köhler M, Moldt D, Rölke H (2000) Renew—the reference net workshop. In: Tool demonstrations, 21st international conference on application and theory of Petri nets. Computer Science Department, Aarhus

Langley P, Hirsh H (1999) User modeling in adaptive interfaces. In: Proceedings of user modeling, Banff

Lavie T, Meyer J (2010) Benefits and costs of adaptive user interfaces. Int J Hum Comput Stud 68(8):508–524

Miñón R, Abascal J (2012) Supportive adaptive user interfaces inside and outside the home. In: Proccedings of user modeling, adaption and personalization workshop, Girona

Navarre D, Palanque P, Basnyat S (2008a) A formal approach for user interaction reconfiguration of safety critical interactive systems. In: Computer safety, reliability, and security. Tyne

Navarre D, Palanque P, Ladry JF, Basnyat S (2008b) An architecture and a formal description technique for the design and implementation of reconfigurable user interfaces. In: Interactive systems. Design, specification, and verification, Kingston

Nieuwenhuis R, Oliveras A, Tinelli C (2006) Solving sat and sat modulo theories: from an abstract davis-putnam-logemann-loveland procedure to dpll (t). J ACM 53(6):937–977

Parasuraman R, Sheridan T, Wickens C (2000) A model for types and levels of human interaction with automation. IEEE Trans Syst Man Cybern Syst Hum 30(3):286–297

Paternò F (2004) Concurtasktrees: an engineered notation for task models. In: The handbook of task analysis for human-computer interaction. CRC Press, Boca Raton

Paternò F (2012) Model-based design and evaluation of interactive applications. Springer, Berlin

Pierce BC (1991) Basic category theory for computer scientists. MIT press, Cambridge

Pleuss A, Wollny S, Botterweck G (2013) Model-driven development and evolution of customized user interfaces. In: Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems, London

Reinecke K, Bernstein A (2011) Improving performance, perceived usability, and aesthetics with culturally adaptive user interfaces. ACM Transact Comput Hum Interact 18(2):1–29

Schürr A, Westfechtel B (1992) Graph grammars and graph rewriting systems. Technical report AIB 92-15, RWTH Aachen, Aachen

Stückrath J, Weyers B (2014a) 2014-01: lattice-extended coloured petri net rewriting for adaptable user interface models. Technical report, University of Duisburg-Essen, Duisburg

Stückrath J, Weyers B (2014b) Lattice-extended cpn rewriting for adaptable ui models. In: Proceedings of GT-VMT 2014 workshop, Grenoble

Weber M, Kindler E (2003) The Petri net markup language. In: Petri net technology for communication-based systems. Springer, Berlin

Weyers B (2012) Reconfiguration of user interface models for monitoring and control of human-computer systems. Dr, Hut, Munich

Weyers B (2015) Fill: formal description of executable and reconfigurable models of interactive systems. In: FoMHCI workshop in conjunction with EICS 2015, Aachen

Weyers B, Borisov N, Luther W (2014) Creation of adaptive user interfaces through reconfiguration of user interface models using an algorithmic rule generation approach. Int J Adv Intell Syst 7(1&2):302–336

Weyers B, Burkolter D, Kluge A, Luther W (2012) Formal modeling and reconfiguration of user interfaces for reduction of human error in failure handling of complex systems. Int J Hum Comput Interact 28(10):646–665

Weyers B, Luther W (2010) Formal modeling and reconfiguration of user interfaces. In: Proceedings of the international conference of the Chilean computer science society, Antofagasta

Weyers B, Luther W, Baloian N (2011) Interface creation and redesign techniques in collaborative learning scenarios. Future Gener Comput Syst 27(1):127–138