Reconfiguration of User Interface Models for Monitoring and Control of Human-Computer Systems

Von der Fakultät für Ingenieurwissenschaften

der Universität Duisburg-Essen

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

genehmigte Dissertation

von

Benjamin Weyers

aus Essen

Referent: Univ.-Prof. Dr. rer. nat. Wolfram Luther, Universität Duisburg-Essen

Korreferent: Univ.-Prof. Dr.-Ing. Dirk Söffker, Universität Duisburg-Essen

Korreferent: Assoc. Prof. Dr. rer. nat. Nelson Baloian, Universidad de Chile

Tag der mündlichen Prüfung: 19.12.2011

Für Ann-Kathrin

Danksagung

Die vorliegende Arbeit entstand während meiner Promotion als Stipendiat der Studienstiftung des deutschen Volkes am Lehrstuhl für Computergraphik und wissenschaftliches Rechnen an der Universität Duisburg-Essen.

Mein aufrichtiger Dank gilt in erster Linie meinem Doktorvater Herrn Prof. Dr. Wolfram Luther, der diese Arbeit angeregt und wissenschaftlich begleitet hat. Trotz vielerlei anderer Verpflichtungen hat er meine Arbeit stets unterstützt und fand immer Zeit für Rat und Anregungen. Vor Allem bin ich ihm für seinen offenen Umgang und für die vielen fruchtbaren Diskussionen und Gespräche dankbar.

Auch möchte ich mich an dieser Stelle bei meinen Korreferenten Herrn Prof. Dr. Nelson Baloian und Herrn Prof. Dr.-Ing. Dirk Söffker für die vielen Anregungen und wissenschaftlichen Kooperationen bedanken, die mich immer wieder aufs Neue motiviert und inspiriert haben.

Die Arbeitsatmosphäre am Lehrstuhl für Computergraphik und wissenschaftliches Rechnen werde ich immer in guter Erinnerung behalten. Ich danke daher allen Kollegen für das besonders gute und freundschaftliche Arbeitsklima, sowie für die vielen fachlichen und nicht-fachlichen Gespräche. Insbesondere möchte ich Herrn Dr. Thomas Pilz und Herrn Dr. Roger Cuypers danken, die nicht nur Kollegen waren, sondern Freunde geworden sind. Auch möchte ich mich bei den zahlreichen Studenten bedanken, die ich im Rahmen ihrer Abschlussarbeiten betreuen durfte. Dabei möchte ich mich insbesondere bei Herrn Dipl.-Inform. Armin Strobl, Herrn Dipl.-Inform. Jan Stückrath, Frau Dipl.-Inform. Sema Kanat, Frau Dipl.-Inform. Dudu Canpolat, Herrn Dipl.-Inform. Cüneyt Altunok, Herrn Dipl.-Inform. Alexander Emeljanov, Frau Dipl.-Inform. Yu Liu, Herrn Dipl.-Inform. Nikolaj Borisov, Frau Hanna Berdys, B.Sc. und Herrn Patrick Berdys, B.Sc. bedanken.

Ganz herzlich möchte ich mich bei meiner Familie und meinen Freunden für ihre Ermutigungen und den Halt bedanken, den sie mir immer wieder gegeben haben. Vor Allem möchte ich mich allerdings bei meinen Eltern, Annegret und Wolfgang, für ihre immer währende Unterstützung während meines Studiums und meiner Promotion bedanken, die im Wesentlichen zum Entstehen dieser Arbeit beigetragen hat.

Mein ganz besonderer Dank gebührt allerdings meiner Ehefrau Ann-Kathrin. Sie hat es nicht nur immer wieder geschafft mich in schweren Situationen aufzubauen, sondern war auch jederzeit für mich da. Sie hat mir in vielen Momenten den nötigen Rückhalt gegeben und stand stets hinter mir und meinen Entscheidungen. Sie fand immer die passenden Worte und öffnete mir den Blick für andere Sichtweisen. Ich danke Dir von ganzem Herzen.

Duisburg, im Dezember 2011 Benjamin Weyers

Contents

1. Introduction

In recent decades, computer science has become increasingly established as the leading field in the development of technologies in interdisciplinary scientific projects. Various examples can be found in communication technology, mechanical engineering, medicine, economics and other disciplines. Having arisen from early data processing in the 1950s, today computer science concentrates on investigating abstract description languages often based on mathematical concepts and working towards computer-based implementations as tools for investigation or as final products. The resulting modeling approaches seek to offer a universal platform for various research disciplines, to the benefit of all participants. Thus, developing abstract modeling strategies for use in digital systems, computer science offers a common universal basis in interdisciplinary research projects for communication, modeling, and development, as well as tool support for scientific work.

Motivation

With the development of electric systems and highly integrated processing units, technical systems are becoming ever more complex, as are the control and monitoring tasks to be conducted by human users. For this reason, the development of human-computer interfaces has become the focus of research seeking to reduce the complexity of technical systems by implementing user interfaces that link psychological aspects, on the one hand, with technical requirements, on the other. This highly interdisciplinary research area is human-computer interaction. Its aim is to develop theories, concepts, and tools that offer user interfaces for successful control of complex technical systems to simplify everyday life.

The field of human-computer interaction involves two scientific areas: *psychology*, which investigates humans, their behavior, and how they understand the world, and *engineering*, which develops and investigates complex technical systems. Computer science concerns itself primarily with modeling methods to describe and investigate systems in executable fashion. Thus, computer science supports (formal, mathematical) modeling methods that can be run on computers for simulation, investigation, validation, and verification paired with tool support for applied modeling by psychologists and/or engineers. In this context, computer science can support methods and tools for modeling systems from the perspective of psychology as well as engineering. Furthermore, computer sciences can use results from both scientific areas to enhance success in human-computer interaction and offer a formal basis for research on merging different worlds to create efficient and effective interactive human-computer systems. In this context, engineering offers various application scenarios, where psychology supports the creation of mental models, statistical evaluation approaches, and cognitive concepts like situation awareness, etc.

This visionary view of future work in human-computer interaction motivates a modeling approach for user interfaces that is, on the one hand, suitable for visual and expert-based modeling and, on the other, formally defined to offer a solid basis for future integration of

related psychological and engineering models as well as to make models directly executable on computers. In the end, this approach reduces the gap between modeling and implementation. Thus, this executable, computer-based communication platform enables the combination of concepts from psychology and engineering.

Furthermore, both psychology and engineering are interested in building models that simplify the real world to the degree that it can be understood by researchers and becomes easier to investigate. Computer science offers a huge set of modeling languages and approaches based on various concepts and ideas, always with the goal of using computers to execute and investigate certain models. In this context, overlapping concerns are identified in building models of the human and/or the technical systems involved, which also argue for developing and implementing a solid computer-based and executable language for integrating models into user interface creation processes and implementations.

Goals

The goal of this dissertation is to develop a formal and visual modeling language to describe human-computer interaction embedded as an executable model of user interface design. This formal approach will be extended by formal transformation concepts to offer options for formal reconfiguration of user interfaces to make formally modeled user interfaces more flexible and adaptable. Thus, they can be reconfigured in accordance with psychological research integrating implicit psychological models to formal user interfaces. One example of this is reconfiguring a user interface to reflect the user's mental model of a machine. On the other hand, results from engineering research can be easily introduced into formal user interface models and thus into the interaction of the human user with the machine. For instance, the interaction processes between humans and technical systems that target specific goals can be embedded in the interaction model of a user interface. Thus, this dissertation describes the development of a flexible approach to formal user interface modeling, execution, and reconfiguration. This flexibility will allow future research to extend the current work by applying such concepts as the easy introduction of user interface adaption into an existing user interface model. Such extensions can serve as a common platform integrating elements from both psychology and engineering.

Various concepts can be mentioned in this context. First, task models, which are familiar from human-computer interaction research, play a central role in identifying and describing the tasks a user tries to solve with a given (technical) system. Next, process and dialog modeling approaches should also be integratable into the formalism to be developed. From the engineering perspectives of automation and system control, the formalism needs to be flexible enough to introduce structures into the user interface model for the automation of certain control and monitoring tasks; it also needs to be able to embed concepts from psychology for the modeling of cognitive architectures and human behavior. Here, it is desirable to develop formal approaches for modeling mental models or use concepts in which the user embeds his mental model in the formally modeled user interface through reconfiguration. Furthermore, it should be possible to introduce structures into the user interface model to identify and avoid errors in interaction. It must be possible to validate and verify the resulting user interface model on the basis of mathematical and algorithmic concepts, as well as to simulate and execute it for use in real life. In conclusion, the formalism to be developed has to offer a solid basis for modeling data processing between a human user and a technical system and to make the resulting user interface reconfigurable at the same time. Furthermore, it should offer formalization abilities for building

hybrid models by introducing different formalization approaches to the interaction model of a formal user interface based on data-driven communication between these models. This makes the formalism more flexible for future extensions and will prepare it for easier integration of formal models from other disciplines.

Resulting from these requirements, a set of working packages can be inferred such as follows:

1. Design of an architecture that subdivides a user interface into conceptual parts such as its suitability for formal modeling of human-computer interaction. Here, it is important to distinguish the functional or behavioral part of a user interface from its outward appearance. Furthermore, the role of the system to be controlled should be specified.

2. Development of a visual language for modeling user interfaces based on this architecture with a view to formal simulation, reconfiguration, and validation and to offering a broad basis for the inclusion of other modeling approaches and concepts, including the ability to build hybrid models. To this end, the visual modeling language should be sustained by a well-known and highly developed formalism that also supports formal semantics for the newly introduced formalism, as well as tools and a vital research community.

3. Development of formal reconfiguration of user interfaces based on the developed formalism. The reconfiguration approach must also be fully formal to prevent problems arising from applying semi-formal transformation approaches to formal and verifiable user interface models. In this way, both aspects are fully formal: (a) the user interface and (b) its reconfiguration. This reconfiguration approach should enable the introduction of further modeling concepts directly into formal user interface models. For instance, one task will be to embed in the user interface model the user's mental model (or parts of it, such as a task model) as a representation of the system to be controlled. The goal of doing so is to reduce errors in interaction. Furthermore, computer-driven reconfiguration should also be possible in order to offer automatic generation of reconfiguration rules and their application to the user interface for implementing adaptive user interfaces as a topic of future work. This goal is similar to another: the introduction of automation concepts to user interface models implementing interaction processes in the user interface model, as well as elements of user help and guidance. In brief, the formal reconfiguration approach should make the formal modeling language flexible enough to meet the above set of requirements concerning hybrid modeling and the transformation of user interfaces' behavior in general.

4. Implementation of a framework offering tools and components for visual modeling, simulation, and reconfiguration of formal user interfaces, as well as offering well-defined software interfaces for further extensions. The visual modeling component should be implemented first and foremost for generating user interfaces on the basis of the formal approach briefly discussed above. The simulation should be able to run the modeled user interface by combining a simulator for the formal representation of the visual modeling language with an interface to the system implementation, which should be controlled through the user interface. Still, the system to be controlled is not part of the investigation and implementation conducted for this dissertation. As will be shown below, the system is initially treated only as a black box represented to the outside world by its system interface (a set of system values). The reconfiguration module of the framework should offer an implementation of formal transformation systems paired with the data structures necessary for describing

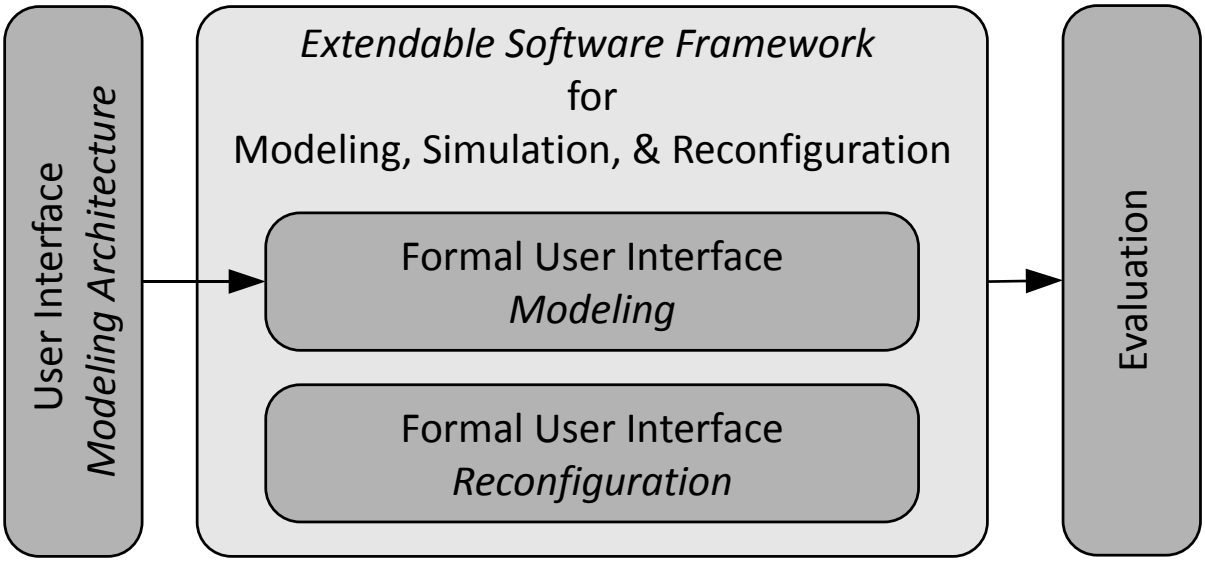


Figure 1.1.: Working packages

transformations. It should also offer an interactive interface that implements a handy way for human users to apply certain reconfiguration operations to the user interface, such as integrating mental models into the user interface model in an interactive and iterative process. Furthermore, the framework should offer an open architecture for further extensions based on well-defined software interfaces; these might include the analysis of human-computer interaction, automatic generation of transformations and reconfigurations, formal verification tools, and data-driven analysis of interaction.

5. Evaluation of the developed approach concerning its application in real work scenarios for identifying its use in various fields and its relevance for future research on integrating psychological models and models from engineering into human-computer interfaces. In the context of this work, the evaluation should determine the extent to which mental models can be integrated into formal user interface models and the extent to which this integration influences interaction and the number of errors made during interaction.

Figure 1.1 shows the different working packages. Based on an architecture for modeling user interfaces, a formal modeling language should be developed along with a suitable formal approach to reconfiguration of user interfaces. These formalisms will be embedded in an expendable software framework for modeling, simulation, and reconfiguration, which will then be evaluated to determine the influence of the approach on human-computer interaction.

Organization

First, a short review of various research areas will be provided. It will identify areas and concepts in psychology and computer science relevant to the modeling of user interfaces and human-computer interaction (Chapter 2). Thus, the survey of the relevant literature will embrace cognitive psychology as it relates to modeling, as well as the classification and identification of human error as relevant aspect in human-computer interaction for safety critical application scenarios. Aspects of automation and its influence on human-computer interaction will also be

of interest. There are various approaches to task and dialog modeling for describing tasks to be fulfilled using a certain system, on the one hand, and, on the other, the extent to which dialogs between human and computer can be modeled and developed. Concerning the reconfiguration of user interfaces, various studies have been published and are included in the survey. A short preview of future work including an overview of work in the areas of multimodal and multi-user interfaces concludes Chapter 2.

Based on this introduction to the field of human-computer interaction and its interdisciplinary environment, the developed formal approach to modeling user interfaces will be introduced. It is based on a visual language paired with formal transformation to reference nets, a special type of Petri nets (Chapter 3). The visual language integrates concepts of business process modeling. These processes will be subsumed in a container called *interaction logic*, which connects to physical elements of the user interface, called the *physical representation*, on the one hand, and with a well-defined interface of the system to be controlled, on the other. To avoid complex definitions of semantics, this modeling language will be equipped with a transformation algorithm that transforms it to reference nets offering formal syntax and semantics. Petri nets are a well-known family of formal languages for modeling non-deterministic processes also involving complex data types and continuous time concepts with an active and well-organized research community offering a broad formal background supporting the formalism and its use. Furthermore, reference nets will be shown as highly suitable for integrating different formalization to build the hybrid models necessary for integrating models from various scientific approaches. Nevertheless, Petri nets in general and reference nets in particular are formalisms that are equipped with a stable tool for modeling and simulation, which is further enhanced and developed [150].

As a central element in platforms combining psychological concepts and technical systems, formal reconfiguration of user interfaces is of great importance. This approach offers transformation to various meta-models and third-party implementations without leaving the formal surroundings of the developed modeling language except for the use of hybrid modeling approaches (Chapter 4). This will be achieved through the introduction of formal graph transformation systems developed primarily in theoretical computer science and often used as a tool for defining semantics for graph-based languages. Here, graph transformation systems will be used to apply reconfiguration rules to the reference net-based model of a user interface in a formal manner, not least because of its solid theoretical foundation.

Modeling approaches in computer science are only as helpful as the tools they support. Therefore, after introducing and defining formal modeling languages and their reconfiguration, the UIEditor framework will be described. This is a software framework implementing modules and software interfaces for visual modeling, simulation, and reconfiguration of user interfaces (Chapter 5). Various open source libraries were used for the visual modeling of graph-based structures. The simulation engine called *Renew*, which can be used for the modeling and simulation of reference nets, is included in the implementation and enables connection to third-party models during runtime by building executable hybrid interaction models. Furthermore, a graph transformation system has been implemented that uses XML-based file formats to describe transformation rules and reference nets based on the so-called double pushout approach.

Chapters 6 and 7 introduce three studies showing that (a) it is possible to transform mental models into formally modeled user interfaces through reconfiguration and (b) reconfiguration reduces human error in interaction. The first of these concepts was investigated in the context of computer-supported cooperative learning. Here, it was possible to show that reconfiguration

promotes success in the learning of cryptographic algorithms. Chapter 7 describes a study in which two processes had to be monitored and controlled by users who were able to reconfigure their user interfaces in accordance with their needs and their understanding of the processes. The control group was not able to use reconfiguration. In these studies, it was possible to show that reconfiguration of the user interface significantly influenced on the number of errors made by the individual user in handling malfunctions in the process.

Chapter 8 offers a broader view of future work, identifying various areas for further development, introducing concepts of interaction analysis. The dissertation concludes with Chapter 9, which looks beyond interaction analysis and automatic reconfiguration to broader implications for future research.

2. Human-Computer Interaction

Human-computer interaction (HCI) is a well-known research field in computer science arising in the 1960s [177]. The study of HCI provides the background for this dissertation, which focuses primarily on the conception of terms and on positioning of this work in a wider context of HCI research. The motivation for this work can be shown by describing related literature and highlighting relevant historical aspects of and developments in HCI. Furthermore, the role of cognitive psychology in context of HCI and of this work will be explored.

2.1. Introduction

Beginning in the 1980s, interest in the area of HCI research increased dramatically with the advent of graphical user interfaces on the first personal computers like Apple's Lisa and Macintosh, and IBM's PC [231]. In HCI research, the focus of interest changes over time [247, 254]. Up till now, the main interest has been to find the answer to the question how to build or create a dialog between human and computer such that the user can solve a task effectively and efficiently without making too many mistakes. For instance, in the early 1980s Shneiderman (1982) coined the term *direct manipulation* in his article "The future of interactive systems and the emergence of direct manipulation" [253] to describe a concept for HCI that still endures today. There, he characterizes the concept of direct manipulation as the "... visibility of the object of interest, rapid reversible actions and replacement of complex command language syntax by direct manipulation of the object of interest" [253, p. 246]. Three years later, Norman and his colleagues investigated this concept of direct manipulation from the perspective of cognitive science in their article "Direct manipulation interfaces" [117], identifying the pros and cons of HCI from this perspective. On the basis of these examples, it is clear that HCI research always combines various research areas. This is underscored by the various definitions of the term HCI and the differing areas of research it includes.

The following definitions of the term HCI provide a more detailed view of this research area from the point of view of various authors, beginning with Dix, Finlay, Abowd, and Bealy who describe HCI as follows:

> As computer use became more widespread, an increasing number of researchers specialized in studying the interaction between people and computers, concerning themselves with the physical, psychological and theoretical aspects of this process. This research originally went under the name man-machine interaction, but this became HCI in recognition of the particular interest in computers and the composition of the user population! [64, p. 3]

Dix et al. [64] identify three central aspects of HCI: (a) the physical aspect of interaction of user and computer, (b) the psychological aspect, which concentrates on the user, and (c) the theoretical background, on which the whole interaction process is based. This definition somehow lacks specificity. The last point in particular should be more clearly defined. Faulkner

[88] also divides HCI research into three different areas, but gives clearer insight into its basic concepts and tools by characterizing the aim of HCI research:

> The aim [of HCI] is to create computer applications that will make users more efficient than they would be if they performed their tasks with an equivalent manual system. The last point is very important, since all too often computerized applications are produced that do not make the user's task easier and more satisfying, nor do they save time. [88, p. 2]

According to Faulkner, the aim of HCI research is the development of models, methods and concepts for generating user interfaces that make systems more usable in that they are more efficient and make fewer errors in solving given tasks. Faulkner goes on to sketch a way for HCI research to reach this goal by combining results from various areas of research:

> ... [HCI] needs to gain its inputs from many other associated areas of study because it has to understand ... the computer system, the human user and the task the user is performing. [88, p. 3]

Here, Faulkner characterizes three areas of investigation: (a) the computer systems, (b) the human user and (c) the task to be completed by the system to be accessed via the user interface under construction. This characterization differs from that given by Dix et al. It is more precise in the sense that it breaks down the whole research area of HCI to three subjects of investigation.

Preece goes a step further in her perspective of HCI research. She also takes into account the environment as a necessary subject of HCI research, ending up with four major areas of investigation:

> The knowledge necessary for this [HCI study] comes from understanding how users interact with computer systems in particular work environments. This interaction consists of four components:
>
> - the *user*
> - who has to do a particular *task* or *job*
> - in a particular *context*
> - using a *computer system.* [219, p. 12]

Preece tries to combine these four aspects in one term to describe the successful creation of a user interface: *Usability.* Preece characterizes a usable user interface (in other words a user interface with high usability) as a user interface that fulfills the following four requirements:

> The goals of HCI are to develop and improve systems that include computers so that users can carry out their tasks:
>
> - *safely* (especially in critical systems like air traffic control)
> - *effectively*
> - *efficiently*, and
> - *enjoyably.*
>
> These aspects are collectively known as *usability.* [219, p. 14]

Preece defines usable user interfaces as safe based on their use in critical situations. This kind of situation can be characterized by such factors as a high level of stress and a short time period for decision-making in situations where many disturbing stimuli from the environment distract the user. Here, different interfaces are subject to different requirements if they are to meet Preece's requirement of safeness. Thus, in the context of safeness, usability has to be determined depending on the type of user interface, the situation, and the user. From the point of view of haptic interfaces, like a control stick in an airplane cockpit, different requirements arise concerning usability than with a visual interface as in cases of process controlling, for instance, in a steel factory.

Furthermore, interaction should be effective and efficient in the context of the task to be solved using the user interface. Here, *effective* means that a given task can be fulfilled using a given user interface, where *efficiency* in this context means that the interface should enable the task to be completed quickly and with as few errors and problems as possible. The last point, enjoyableness, seeks to convey a kind of 'fun factor' provided by using the interface. Here, Preece subsumes factors required for the user's motivation to use the user interface. For instance, two user interfaces that are identically safe, effective, and efficient but differ in their 'fun factor' affect their perceived usability only because the user's motivation differs.

In the book *Designing the user interface* [254], Shneiderman and Plaisant characterize HCI in a more complex way. They describe the criteria for the quality of a user interface concerning the aspects of "usability, universality, and usefulness" [254, p. 31] in context of the ISO norm 9241[1] [254, p. 32] as a set of requirements:

1. *Time to learn.* How long does it take for typical members of the user community to learn to use the actions relevant to a set of tasks?
2. *Speed of performance.* How long does it take to carry out the benchmark tasks?
3. *Rate of errors by users.* How many and what kind of errors do people make in carrying out the benchmark tasks? Although time to make and correct errors might be incorporated into the speed of performance, error handling is such a critical component of interface usage that it deserves extensive study.
4. *Retention over time.* How well do users maintain their knowledge after an hour, a day, or a week? Retention may be linked closely to time to learn, and frequency of use plays an important role.
5. *Subjective satisfaction.* How much did users like using various aspects of the interface? The answers can be ascertained by interviews or by written surveys that include satisfaction scales and space for free-form comments.

The way in which Schneiderman and Plaisant outline the basic requirements of user interface development also defines a framework for testing the quality of a given user interface. By applying a set of benchmark tasks, the developer of a user interface can determine the quality of a given user interface design. Again, the human user is the focus, which seems to be a central theme in HCI research.

It can be concluded from the work of the above authors, as well as that of Sears and Jacko [247]; Sharp, Rogers and Preece [252]; Rosson and Carroll [238]; and Card, Moran, and Newell [40], that HCI research embraces psychology, technology and sociology. At the same time, of

[1] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38009

course, it includes computer science, which can be seen in the work of Palanque and Paternó [204], who describe the use of formal methods in HCI.

In order to go into more detail, it is of interest to narrow the investigation to only central aspects. The goal is to create methods and tools to (a) model the user interface on various formal levels, (b) describe reconfiguration techniques based on this formalization and (c) to define ways of formalizing the results of differing types of HCI research, so that bridges can be built between formal studies in computer science and semi-formal or informal concepts from HCI and cognitive psychology research. The following sections will specify earlier work that contributes to the realization of these goals.

2.2. Cognitive Psychology and Modeling

Various methods for modeling the user, the user interface, the task and the interaction itself have been developed and investigated in HCI research. Concepts like hierarchical task trees, cognitive models and architectures are only a few examples of results from cognitive and HCI research. This section will make the user the focus of investigation in order to identify possible cognitive psychology approaches for further investigation and to build a basis for describing the connection between formal modeling and cognition. Before going further into detail, the term *cognition* should be briefly defined.

In psychology, cognition is usually defined as a *process of the mind*. Thus, *cognition* describes the how humans think about, remember, perceive and learn information. Lachman, Lachman and Butterfield [154] introduced an analogy to computers to describe what cognitive psychology is about:

> Computers take a symbolic input, recode it, make decisions about the recoded input, make new expressions from it, store some or all of the input, and give back a symbolic output. By analogy that is what most cognitive psychology is about. It is about how people take in information, how they recode and remember it, how they make decisions, how they transform their internal knowledge states, and how they translate states into behavioral outputs. [154, p. 99]

Parkin [207] cautions that "one must be careful about how far this analogy can be taken" [207, p. 12]. Nonetheless, he states that this analogy is still good enough to sketch out what cognitive psychology is about. For instance, he agrees that the central processing unit of a computer, which combines numbers using specific operations to obtain a result, can be compared to the identical part of the human mind doing the same thing. The same is true for information storage on hard drives for long-term purposes or RAM as an information buffer. The same modules can be found in the human mind as parts of information processing. But, says Parkin, the analogy ends at the point of information representation. The computer represents all information using a binary system, which is not true for the human mind.

In order to understand the role of cognitive psychology in the context of this work, it is useful at this point to revisit the main goal of this work: the formal modeling of human-computer interfaces using a psychological approach to model human cognition. The basic idea is motivated by research results from the investigation of adaptive user interfaces, which will be examined in an upcoming section. Adaptive user interfaces seek to enhance the usability, efficiency and effectiveness of user interfaces by adapting them to users' abilities and approaches to solving

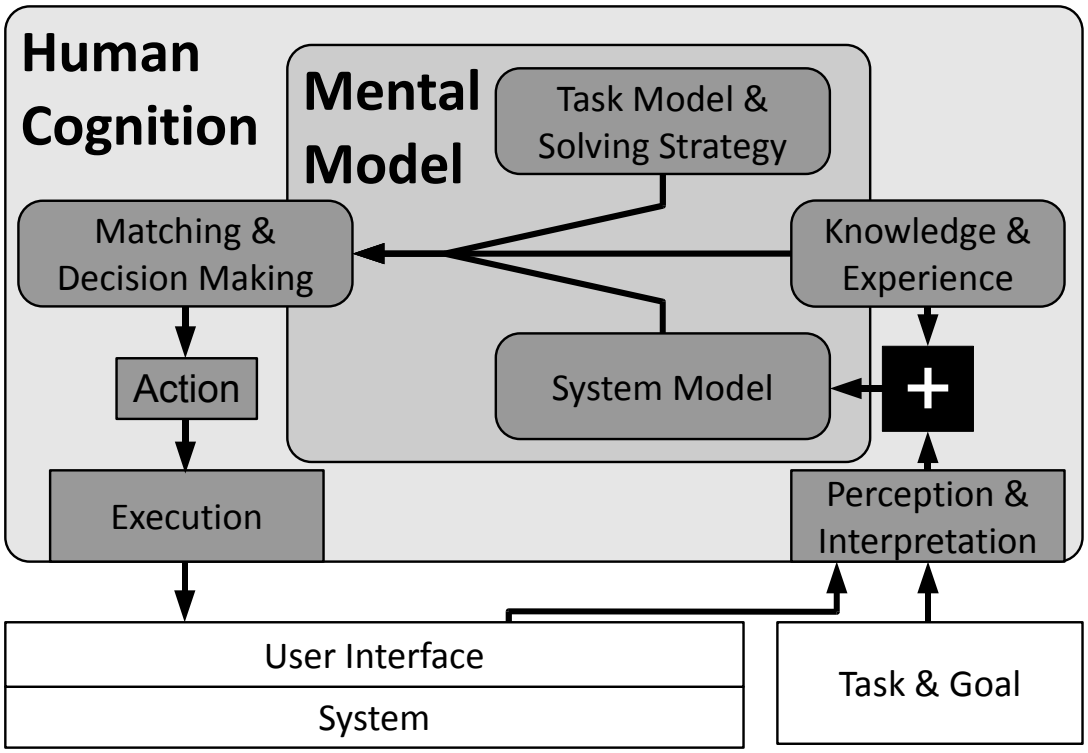


Figure 2.1.: Model of the human cognition for task solving—a brief overview

a given task. In other words, adaptive user interfaces seek to adapt to the cognitive workflow of users who are solving a given task by using a certain tool. Figure 2.1 shows a model of human cognition, providing a deeper understanding of what is meant by the cognitive workflow of information processing and task solving.

Nevertheless, this overview addresses only a few aspects of research into human cognition, focusing on the concept of *mental models*, but relating the computer analogy quoted above to human cognition in specific ways intended to support the formalization introduced in the following chapters.

The concept of mental models seems to be a good point of connection between cognitive psychology and research into adaptive user interfaces. Thus, the structure and occurrence of mental models, which represent the tools and knowledge an individual uses to solve a given task, are highly relevant to the process of adapting user interfaces. A mental model represents the task and the strategy for solving the task using a computer or a machine as tool. The cognitive representation of the function of this tool is a further part of a mental model. An introductory overview of the term *mental model* and related research in the context of HCI is given by Rasmussen [227] and in the context of human factors by Wilson and Rutherford [298]. Various psychological works introduced various views, approaches and concepts due to mental models. The definition given by Rouse and Morris [239] deals with the context of HCI:

> ...[A mental model is a] mechanism whereby humans are able to generate descriptions of system purpose and form, explanations of system functioning and observed system states, and prediction of future system states. [239]

In this sense, a mental model is essential for human cognition to interact with a machine. Thus, a mental model describes how to interpret and how to 'understand' the machine or its

representation in the user interface such that a human can understand the state of the machine, know how it actually works and, in the end, predict future system behavior based on this knowledge. Additionally, the user should be able to affect the state of the machine in order to achieve any new state that might be necessary to perform given tasks.

A mental model can also be seen as a massive repository of long-term memory as described by Carroll and Olson [41] and by Kluwe and Haider [136]. Parkin [207], Anderson [5] and Eysenck [87] offer more detailed information concerning models for organization of human memory as part of human cognition. In general, the term *long-term memory* can be described as a type of memory that offers learned concepts and experience as the basis for processes like decision making or interpretation in human cognition. Tulving [269] described long-term memory by dividing it into two parts: (a) episodic memory, which stores discrete personal events, for instance, what we had for dinner, and (b) semantic memory, which holds knowledge concerning language and the world, for instance, the fact that rain is wet. Furthermore, he [270] adds to this list procedural memory, which holds knowledge about how to do things, for instance, motor skills, like moving one's legs and arms or grasping an object.

As stated before, the mental model is one part of the whole cognitive process. Problems in this process can occur if a mental model is erroneous. An erroneous mental model can be described as an incorrect image of the reality. Errors will occur in user interaction with a machine that is based on an incorrect model. In conclusion, if errors occur, an erroneous mental model may be the reason. Mental models are generally constructed by practicing and training, and in this process, incorrect assumptions to the model can be manifested, as described by Kluwe [136].

A further problem arises if the user interface with which the human user is interacting is not completely applicable to his mental model. For example, the user expects a value to be changed over time (part of the mental model) but its representation is only a static number not visualizing its changing characteristic. The user's mental model of the machine might be correct, but if the user interface does not represent it in a specific and expected way, the user will have trouble matching the perceived information to his mental model and will start to generate wrong predictions concerning the current and future system state, which will result in errors in interaction and in controlling the machine. Here, an adaptive user interface will try to customize the presentation of the machine implemented in the user interface to the specific user's mental model.

Mental models are often described as sets of cues that in turn describe well-defined parts of a bigger work flow or process, such as described by Ericsson and Kintsch [85]. Thus, in this work the behavior of a user interface—also called interaction logic—is modeled as a set of interaction processes that can be directly matched to the cues in the user's mental model. This is a first attempt to bridge the gap between formal modeling in user interface development and the research on mental models through the use of formal methods.

A further way to describe mental models was introduced by Rauterberg, Schluep, and Fjeld [228]. They describe the use of Petri nets to represent mental models in the context of automatic determination of mental models from log files. They distinguish four types of models with differing complexity, where each can differentiated between five different levels in a mental model and include the system and the feedback from the system as mental representation. They developed this method of representing mental models when studying actual interaction of experts with a system to solve a given task, thus determining the cognitive complexity of using the system and interface. This approach to determining cognitive models of human users seems a promising way to provide formal modeling and validation for user interfaces.

Another approach to formal description of cognitive models based on Petri nets was introduced by Werther [281]. He uses colored Petri nets to model the cognitive memory system as resource system defining the boundedness of human information processing. He extends the single modeling aspect by applying formal verification methods of Petri nets to models of human cognition. This ends up in statements concerning errors in such models, as well as assessments on occurring problems in interaction. A holistic approach based on the cognitive architecture introduced by Rasmussen [226] has been described in various works of Gamrad and Söffker [95, 96, 256]. Here, a concept called Situation-Operator-Modeling approach (SOM) describes the whole concept of knowledge and information processing of human cognition, which is transfered to colored Petri nets for verification and machine-based learning. This concept has been successfully used in various application scenarios like drive assistance [93] and arrival management in air traffic control [192].

2.3. Human Error

Preventing human error during interaction with a machine is one main motivation for interdisciplinary research in psychology and HCI. To reduce errors in interaction, the structure and mechanisms of human errors have to be understood before starting to create systems that will prevent them. To this end, many researchers have investigated the occurrence of and reasons for human errors in interaction with machines or other (more general) systems. The following overview of research into human error, while by no means complete, gives a brief introduction to the main aspects and concepts in this field of cognitive psychology.

Endsley [83] introduces the framework modeling *situation awareness* that can be seen as a specified model based on the one shown in Figure 2.1. Situation awareness describes the distribution of a person's attention in a certain situation at a given point in time, as well as his or her understanding of the current activities and expectations as to how the situation will progress. This definition of the term situation awareness is Endsley's attempt to integrate various cognitive activities in handling complex and dynamic situations, starting with sense-based perception (visual or haptic perception, etc.) and including processes for decision making and problem solving [135]. Endsley defines three processes that establish situation awareness shown in Figure 2.2: "... the (1) perception (noticing) of the elements in the environment within a volume of time and space, the (2) comprehension of their meaning, and the (3) projection of their status in the near future" [84, p. 15].

In Endsley's framework modeling situation awareness, a classification of human errors can be specified due to [82] and [84]. In these publications, Endsley differentiates three levels of error in the context of the framework for situation awareness: errors in perception of the situation (level 1), errors in interpretation and understanding of the situation (level 2) and errors in the projection of future states (level 3).

Other works, like that of Hollnagel [113], describe other approaches to the topic of human error. Hollnagel introduces a framework called the Cognitive Reliability and Error Analysis Method (CREAM). CREAM is another more specific approach than that shown in Figure 2.1 to performance and error prediction, for instance, in controlling a nuclear power plant [114]. It classifies errors by separating them into genotypes and phenotypes. The genotype describes the possible cause of an erroneous action, whereas the phenotype describes a concrete occurrence of an erroneous action.

Concerning errors in complex decision making, Wickens and Hollands [292], as well as Dörner

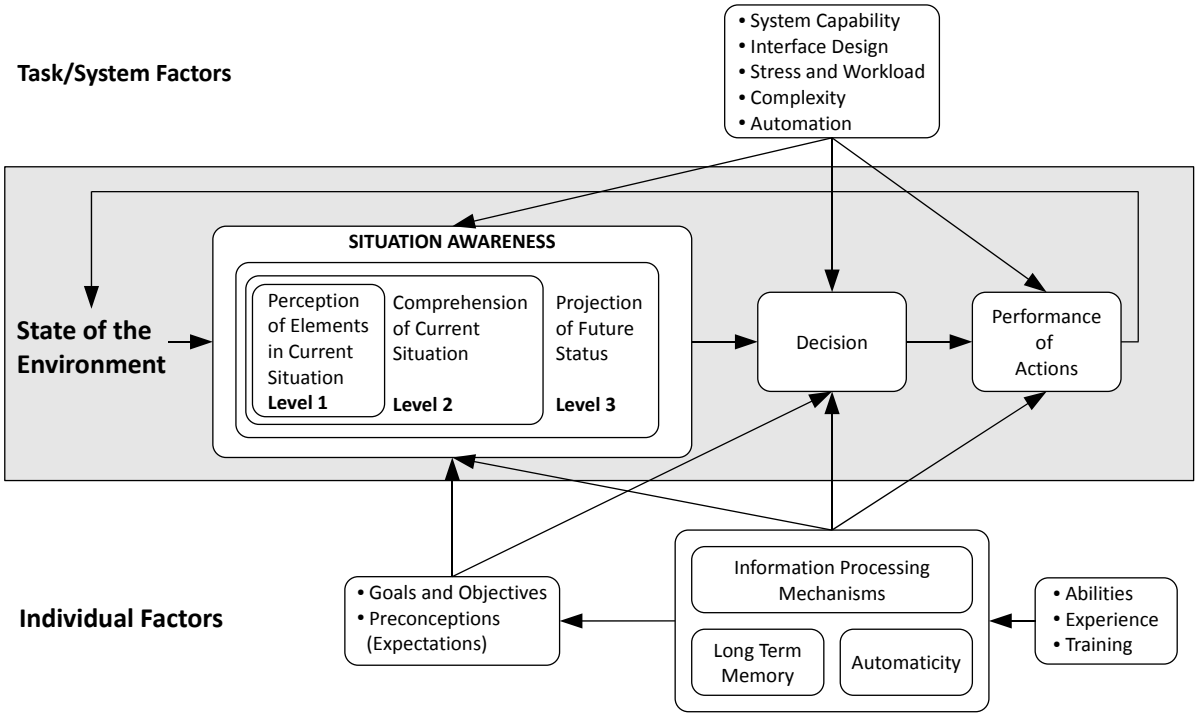


Figure 2.2.: Framework of Situation Awareness (Endsley [83])

[67] describe the problem of *saliency bias* (Dörner's *Lautstärkeprinzip* [67]). This saliency bias describes the problem of stimuli that direct a person's attention to salient values that are less important than they appear. Especially in situations where the person is pressed for time, this bias can have a great influence on the occurrence of human error.

These are only a few basic approaches to the classification of errors and investigation of the cognitive reasons occurring in the context of the cognitive information processing model presented in Figure 2.1. In the 1980s, authors such as Rasmussen [225] and Nowak and Carr [189] sought to identify general classification concepts for human error. More recent authors have examined the topic in more specific contexts, such as medical care [66, 140] and aviation [56, 251].

Many recent studies have examined human error in aviation. An ergonomic perspective, for instance, was described by Edwards [69], who introduced the SHEL model differentiating between four components: S stands for the non-material part of the system to be controlled, also called *software*; H represents the *hardware* of that system, also involving the user interface hardware; E stands for *the environment* in which the further three elements are embedded in a coincidentally related manner; finally, L stands for *liveware* and represents the human user of the system. The relations between these components provide the significant information in the model. For instance, changing the hardware can also influence the interaction between user and system in a positive or negative way. This type of model has been used by accident investigators to identify human error in such situations (cf. [293, p. 29]). Other perspectives (e.g., behavioral, aeromedical, psychosocial, or organizational) and further reading on human error in aviation can be found in the work of Wiegmann and Shappell [293].

Independent of how human error is defined or how the models and frameworks surrounding this topic are constructed, most approaches lack a foundation in formal HCI research. This

work will demonstrate the great value of incorporating human error research into the creation, adaption and reconfiguration of user interfaces in Chapter 7. By providing formal platforms on both sides, computer science research into adaptive user interfaces coupled with research into human error greatly benefits due to the understanding of how human and computer interact and thus significantly improves that interaction.

2.4. The Role of Automation

In the reconfiguration of user interfaces (cp. Chapter 4.2), automation plays a central role as an extension of the adaption of user interfaces. Here, automation refers to specific parts of technical systems that *automate* certain parts of controlling and monitoring of a system, for instance, an autopilot. According to Kluwe [135], automation has a great impact on human-computer or human-machine systems. This impact has been investigated mainly by Parasuraman and his colleagues [205, 206]. As they have shown, the role of automation can be seen in the four main functions of HCI and information processing like that given in Figure 2.1:

Information retrieval How information about the system's state, performance, and achievement of goals is acquired ('Perception & Interpretation' in Figure 2.1).

Information analysis How data is integrated, and how information is evaluated and used for decision making ('Plus' box in Figure 2.1).

Decision and selection of actions Recommendation and selection of actions based on system selection and operator information ('Matching & Decision Making' in Figure 2.1).

Execution of actions by the operator and the system ('Action' and 'Execution' in Figure 2.1).

Automation can be applied to any of these functions depending on the level of control needed (high vs. low risk situations) and the character of the function itself. This kind of automation would be implemented in the 'User Interface and System' box in Figure 2.1. Parasuraman, Sheridan and Wickens [206] describe the role of automation and its functions in the context of air traffic control introduced above. High automation can be applied for information retrieval; in contrast, low automation should be applied for decision making and selection of actions in high risk situations in order to increase the operator's freedom of action.

There are further problems associated with automation. Wickens [291] identified three major problems:

1. **UNDERTRUST**: Malfunctions, for instance unwarranted alarms, lead to a lost of trust in the automation of the controlled system. This undertrust can lead to erroneous intervention by the operator into automated parts of the system.

2. **OVERTRUST**: Too high a degree of trust (overtrust) in automation can lead to such problems as a failure on the part of the operator to identify system malfunctions.

3. **OOTLUF**: The acronym OOTLUF describes a problem arising from automatically generated alternatives to actions applied to the system. It has been shown that action sequences generated by the operator can be better remembered than automatically generated sequences. In the case of processes that were previously automated, the operator needs

more time to generate the correct decisions and apply them to the system. This situation can also be described as a reentering of the operator into the control loop. Therefore, OOTLUF stands for "out of the loop unfamiliarity".

In the context of automation in aviation, a further term should be mentioned: *automation surprises*. This term refers to bad outcomes resulting from poorly implemented automation. Sarter, Woods, and Billings [240, 241, 299] describe problems posed by automation for aviation purposes and the potential for bad outcomes in high risk situations from an engineering perspective.

Similarly, automation resulting from adaption or from the reconfiguration of user interfaces also has to be considered in the context of automation as introduced above. Problems arising from the reconfiguration of parts of the user interface need to be identified in an early stage or avoided altogether by applying findings like those described by Parasuraman. Thus, an environment that embeds the interaction logic of a user interface with formal reconfiguration or adaption offers a handy formal approach to the problem of automation.

2.5. Task and Dialog Modeling

The last box in Figure 2.1 has not yet been discussed in further detail. This is the 'Task Model & Solving Strategy' box. Closely related to the psychological approaches in HCI research introduced above are the formal and semi-formal modeling approaches to dialog and task modeling familiar from classical HCI research. The main motivation here is the modeling of a single, more a general 'standard' user in the context of solving tasks on the basis of hierarchical structures. The outcome of this modeling strategy is a user interface that is customized to the user's needs concerning the strategy for solving given tasks and thereby increasing the effectiveness of using the user interface. Various models and approaches were developed in the past on different levels of task-planning strategies. Dix, Finlay, Abowd, and Beale [64] separate these levels into the following three layers:

1. hierarchical representation of the user's task and goal structure,
2. linguistic and grammatical models, and
3. physical and device-level models.

Two well-known examples of hierarchical task models based on mental processing are GOMS and CCT. GOMS, developed by Card, Moran and Newell [38], is a modeling method based on four different concepts: (G)oals refers to the goals the user tries to achieve using an interactive system where those goals simultaneously represent points in the user's memory; (O)perators refers to the elementary actions the user can or has to perform on the system, for instance, pressing a certain key; (M)ethods are rules governing how goals can be split into subgoals; (S)elections are rules for selecting a certain method, depending on the user and the context, for instance, the current state of the system.

The second approach for using basic psychological research in mental processing and task solving is the (C)ognitive (C)omplexity (T)heory (CCT) introduced by Kieras and Polson [134]. CCT 'inherits' GOMS' goal decomposition concept and enriches it by introducing a system model to enlarge the power of prediction. In contrast to GOMS, CCT builds on production

rules defining a GOMS-like structure of task decomposition. Production rules in CCT are based on if-clauses offering else-branches.

Based on the idea of using production systems, linguistic and grammatical models are further examples of task hierarchy models. The task action grammar developed by Payne and Green [214] is one well-known example of a linguistic approach.

The (K)eystroke (L)evel (M)odel, KLM, is a physical or a device-level model and can be described as a simplified and practical instance of the GOMS family [184]. KLM was developed by Card, Moran and Newell [39] and describes elementary operations that are paired with an execution time. The execution sequence of these elementary operations that results from a task model, for instance, one using GOMS, can be used to predict (time) costs in interaction.

All these early approaches were the first to combine psychological research into human cognition with computer science research into HCI. In their book *The Psychology of Human-Computer Interaction*, Card, Moran and Newell [40] brought the two worlds together as one part of computer science. Many other groups also worked on involving psychological concepts in the modeling of human-computer interfaces, including Anderson et al. [6], Robertson and Black [236], Norman [188], and Young [302].

Besides task modeling and analysis, dialog modeling methods play an important role in HCI research. Here, a *dialog* refers to the exchange of information between human and computer in contrast to task-modeling techniques, which refer to decomposition and problem-solving strategies. This type of model is a departure from the context of cognitively motivated models such as GOMS and CCT. Dialog models in the sense presented in this section are not directly based on cognitive processes but are inspired by them.

The first approach to modeling dialogs between human and computer was the use of state transition networks, which were identified in the late 1960s [186, 208]. Here, a state represents the state of the interactive system, and the transition represents the user's input. This approach has one disadvantage: Every state that could be reached in a dialog had to be modeled explicitly, resulting in a state space explosion. Because of this, alternative modeling languages were developed. One example of an alternative language is Petri nets, where states are represented as markings in a net and not as explicit structures of it. Petri nets were used to specify single- [202] and multi-user system dialogs [203]. Later works by Bastide and Palanque [19, 20] describe an approach to dialog modeling that uses visual languages based on Petri nets and embedded in an object-oriented development process. Other approaches to dialog modeling are combining modeling languages like UML [196] with Petri net-based formalisms [80]. Furthermore, Janssen, Weisbecker and Ziegler [122] introduced an extension of classical Petri nets for dialog modeling called *dialog nets*. In addition to Petri net-based approaches, grammar-based and production system-based approaches were also developed and used, for example, the propositional production system (PPS) developed by Olsen [194].

The main problem in all these approaches is the lack between modeling and implementation of a user interface. A contrary example is the Microsoft® tool Expression Blend® for developing user interfaces [169]. This software is an integrated development tool that combines visual modeling of the user interface with a graph-based modeling component for creating the dialog structure of the user interface. Thus, a closer integration through formalization of modeling and implementation can help in creating a user interface and refining it through methods like reconfiguration as shown by Microsoft's Expression Blend on a visual and semi-formal level.

A formal modeling approach to interaction and dialogs is developed by Navarre et al. [182] introducing a modeling language called ICO. This language is pretty similar to the Petri net-

based modeling language that is subject of investigation in this dissertation. Still, the new approach in this dissertation differs in various aspects from ICO that mainly extends the dissertation of Palanque [201]. The first aspect is the modeling approach the group of Palanque follows. They use a Petri net based formalism, which is extended by introducing programming code as inscriptions to a Petri net-based formalism. This ends up in the problem that the modeler needs programming skill. Furthermore, Petri nets are not intuitive to use for modeling visually. Another aspect is the attended implementation in this dissertation, which is planned as a framework for modeling, simulation and reconfiguration of user interfaces. PetShop[2] [181] as implementation for modeling ICO only offers the simulation of the user interface without direct connection to a system or a formal implementation of reconfiguration. Still, the implementation is executable but the code is not open and thereby makes it impossible to compare other approaches to PetShop. Last but not least the group around Palanque developing ICO did not yet publicized evaluation studies concerning the question in how far formal modeling does influence HCI, what should be the main goal for modeling in HCI.

2.6. Adaptive User Interfaces

In the middle of the 1990s, HCI research began trying to incorporate results from artificial intelligence research in order to build intelligent user interfaces. This led to the first research into adaptive user interfaces. The idea in this field is to create a model of the user such that adaptions for the user interface can be derived by using that model. The user model in this case represents the user's cognitive abilities and his principles of operation, and therefore functions as an information basis for adapting the user interface to the user's abilities. The result of using the adapted user interface is a more efficient and effective interaction between the user and the system.

Before going into detail, the terms *adaptive user interface* and *intelligent user interface* should be properly defined. These terms will be defined by using one person, one machine, one task scenarios. This type of scenario subsumes the case of one user who tries to solve one task by using one user interface. (The single-user scenario will be sufficient for the purposes of definition. Later, the formal modeling of multi-user interaction will be addressed, as a multi-user, multi-interface scenario, which is the focus of current research.) Dietrich et al. [62] explain an adaptive user interface as follows:

> [Adaptive user interfaces] are designed to tailor a system's interactive behavior with consideration of both individual needs of human users and altering conditions within an application environment. [62, p. 13]

Dietrich and his colleagues address two major aspects of adaptive user interfaces: (a) the human user and (b) the environment. Both have individual needs and altering conditions depending on the interactive behavior of the system they must interact with. The behavior of the user interface must be adapted to reflect these needs and conditions. Langley, whose primary research focus is machine learning [156], provides a further definition of an adaptive user interface [158]. He goes a step further by claiming not only that a user interface should be tailored to meet those needs and conditions but that it should actually improve its functionality:

[2] http://www.irit.fr/recherches/ICS/softwares/petshop/

> An adaptive user interface is an interactive software artifact that improves its ability to interact with a user based on partial experience with that user. [158, p. 358]

Here, Langley defines an adaptive user interface in reference to its ability to adapt to the user based on the experience it gains through interacting with that user. The controlled system is not mentioned directly but is implicitly understood as being controlled by the user through the user interface. Both definitions focus on the adaption of the user interface to the user. The term *intelligent user interface* goes a step further, as can be seen in this definition by Maybury:

> Intelligent user interfaces ... are human-machine interfaces that aim to improve the efficiency, effectiveness, and naturalness of human-machine interaction by representing, reasoning, and acting on models of the user, domain, task, discourse, and media (e.g., graphics, natural language, gestures). [166, p. 2]

In this sense, intelligent user interfaces not only involve the user—or a model of the user—but also information about "the domain, task, discourse, and media". Furthermore, Maybury defines the difference between adaptive and intelligent user interfaces, stating that an intelligent interface not only adapts itself to models of the user but also carries out "reasoning" about those models. In the context of this dissertation, developing a formal basis for adapting a user interface to a user is the primary research interest. Therefore, Langley's definition covers all the subjects of investigation necessary for this work: (a) the user and (b) the user interface as a software artifact (extended by a formal basis). As will be shown below, a central part of the user interface will be a formal model of interaction logic describing an executable representation of data processing of input information and data inherited from the system to be controlled. From a modeled system, further important information can be derived, which will be part of a future investigation into error-based reconfiguration of user interfaces where errors in such aspects as interaction can be derived from unwanted or critical system states.

Nevertheless, both types of interfaces use a model of the user in a computer-readable form. From the perspective of cognitive psychology, a formal model of an architecture, like the one shown in Figure 2.1, would be the best solution. Still, in research on adaptive user interface, various approaches have been developed, for instance, by Langley. He publicized works on user modeling from the perspective of machine learning. In [157], he gives an overview of the use of machine learning concepts and approaches in adaptive user interfaces and later [158] describes the concrete use of machine learning for user modeling in adaptive interfaces.

In the late 1990s, besides adaptive user interfaces and user modeling, terms like *software agents* and *intelligent agents* arose to refer to systems that adapt to their users. Jameson [121] describes his work with user-adaptive systems and agents, also extending concepts of adaption to a broader range of research areas and interests.

A third approach to user modeling has been described by Razmerita et al. [229]. They present an ontology-based user modeling architecture with its origins in the IMS LIP specifications from semantic web technologies [47]. Further information is given by Fischer [91], in which he gives an overview of user modeling in HCI research. Both approaches, agent- and ontology-based, are the result of subsequent research based on works like Langley's. Especially the agent-based approach seeks to create multi-user models addressing systems that are applied even outside of the one user, one task, one system scenario.

The second aspect of adaptive user interfaces is the user interface itself or, more specifically, its creation, generation and adaption, which will be dealt with in the next section in the context

of reconfiguration. As many authors have pointed out, the process of developing user interfaces is similar to software engineering processes. The process of creating a user interface begins with a design process that is, for instance, supported by task and dialog modeling, as well as user modeling techniques. However, user interface creation as part of the software modeling process is not explicitly of further interest in this work. Further information on this subject give Dix et al. [64], who present a comprehensive survey of HCI; Shneiderman and Plaisant [254] and Raskin's slightly more recent book [224] as psychology-based approaches to user interface design; and Tidwell's collection of interaction patterns [267], as well as Borcher's pattern-based approach to interaction design [28].

This work focuses on concepts and approaches to automatic or more general computer- or machine-driven creation of user interfaces on the basis of formally or semi-formally described approaches. One well-known example of automatic generation and creation of user interfaces is described by Paternò [209, 210]. Paternò's approach is mainly based on task models, like the ConcurTaskTrees introduced by Puerta [223], which can be seen as extensions to the dialog and task models described above. Here, an initial attempt to generate user interfaces from cognitive models tries to fill the gap between cognitive modeling based on a connection between task- and dialog-modeling techniques and the concrete implementation of user interfaces. The use of model-based approaches is of particular interest in nomadic applications [173]. A nomadic application "... adapts to mobility, ... but retain the richness and control of desktop applications" [193, p. 3]. Well-known examples are mini-applications for mobile phones. Paternò and Santoro [211] describe a more general approach to using descriptions of model-based user interfaces to promote the use of one model of a user interface on various platforms, even in the context of model-based and automatic generation of user interfaces.

Janssen, Weisbecker and Ziegler [122] describe the generation of user interfaces from data models paired with a dialog model given as dialog net specification. These are an example of the necessity of a well-defined formal approach to modeling user interfaces as a counterpart to formal modeling methods on a higher and more abstract level. Offering formal methods on both levels of modeling helps to create highly usable interfaces. A further example of modeling on an abstract level is is given by Eisenstein and Puerta, who describe automated user-interface design using an adaptive algorithm [79]. These approaches motivated adaption mainly from the perspective of the user, who uses the interface to control a machine. Another, newer perspective is to develop a machine that offers services to the user as an agent. This new perspective on user interface creation and adaption is a very interesting new facet of HCI research, especially in the context of formal modeling and reconfiguration of user interfaces. Therefore, this work can be seen as an example of the close relationship between automated user-interface creation and concepts of adaption as approaches to user interface creation as a continuous process rather than as an application of a given model to a final and static user interface.

2.7. Reconfiguration

Reconfiguration of user interfaces can be approached from different directions, such as a tool-driven approach that seeks to develop tool support for various concepts and ideas or to develop formal models and languages in a given context. A possible approach to the topic of reconfiguration of user interfaces is that of extending an existing framework, as shown by Stanciulescu [260]. He describes how information systems can be made flexible by extending a task model described in UsiXML [273].

Navarre, Palanque and Basnyat developed tool-supported reconfiguration of user interaction in safety-critical interactive systems [179] based on an architecture they published together with Ladry [183]. They investigated risk situations in airplane steering to identify possible reconfigurations of the cockpit's instruments. The result of this investigation was an expert dialog model for interaction with the cockpit of an Airbus A 320. The entire study was based on an architecture for reconfigurable user interfaces described in a Petri net-based modeling language extended by a reconfiguration manager.

Rosis, Pizzutilo and Berardina [55] describe an approach for synthesizing user interfaces from formally described interaction based on Petri nets. They describe a tool support for modeling user interfaces with the paired formalism of Petri nets and the keystroke level model developed by Card, Moran and Newell [39] called *XDM* (context-sensitive dialog modeling). This combination also introduces a validation strategy into the modeling process.

Reconfiguration is also a well-known term in software engineering and can be described as the process of making changes to software at run-time as described in [276]. Here, Warren, Sun, Krishnamohan and Weerasinghe introduce a framework called OpenRec that offers an open reconfiguration management infrastructure. This system has been extended by a verification mechanism for deciding whether a suggested reconfiguration should be applied to a system or not. This verification mechanism is based on a formal modeling language called ALLOY, which offers a formal notation to specify systems and associated constraints based on first order logic. Another example of reconfiguration in software engineering is that of Wermelinger, Lopes and Fiadeiro [280], who define a high-level language to describe architectures for applying changes to a configuration, such as adding, removing or substituting components. Also, Cansado, Canal, Salaün and Cubo [36] describe a formal framework for unifying the behavioral adaption and structural reconfiguration of software components.

These examples show the close relationship between reconfiguration in various fields of research and formal methods, including formal verification approaches. This forms the basis for developing a formal modeling language based on a well-known formalism offering a broad palette of tools and theoretical background. Such a formal approach also makes sense in the context of computer-based or intelligent adaption of user interfaces. A formal modeling method results in access to all the aspects of reconfiguration, user modeling and cognitive research described above. Such a technology has not been investigated in earlier research, which is why it seems necessary to do now.

2.8. Multimodal and Multi-User Interfaces

Up to now, this discussion has introduced related studies that form the basis for the concepts of formalism described in the sections below. Still, this survey would be incomplete without taking into consideration future work that will focus on modeling formal user interfaces and reconfiguration in multi-modal and multi-user scenarios. Such an extension would widen the one-dimensional view presented here (the one user, one task, one system, unimodal user interface) to embrace a multi-dimensional approach. Another goal is to extend the nomenclature concerning studies in these newer research fields. Nomenclature, however, changes over time in specific research communities and areas. This is also true for HCI research. In the above sections, the focus has been mainly on older studies that form the basis for current research but are not completely up to date in HCI literature, especially related to newer work on multi-user and multimodal user interfaces. Thus, the best indicator of up-to-date nomenclature is pub-

lished in studies on multimodal and multi-user interaction because these research areas have been developed in the time period since 1995.

Before starting to describe the various facets of multimodal and multi-user interface research, the terms *multimodal* and *multi-user* should be defined more clearly. Oviatt [200] identifies and substantiates ten myths of multimodality and gives a brief explanation of a multimodal system:

> Multimodal systems process combined natural input modes—such as speech, pen, touch, hand gestures, eye gaze, and head and body movements—in a coordinated manner with multimedia system output. [200, p. 74]

Thus, multimodality addresses a combination of various interaction channels available to a human user, like the haptic channel for gestures and pen interaction, vision and acoustics for perception of multimedia output, and the aural channel for speech input. These channels form the basis of various research interests that concentrate on the investigation of one specific modality and how they combine with others. Some examples can be found in investigations into gesture-based interaction. Ou et al. [199] present a system that integrates gesture-based input in live video for supporting physical tasks in a collaborative way. In medical contexts, for example, gesture recognition enriches the user's potential to communicate with the system. The main aim is to offer more natural and effective communication between user and machine. A further example of gesture-based multimodal interaction comes from research on collaborative modeling, which also addresses multi-user aspects. Zurita, Baloian and Bytelman [304] describe a tool called MCPresenter, which can be used for mobile modeling in the field supporting gesture-based interaction to enrich communication on small mobile devices.

In the context of mobile applications and devices, other studies use mobile devices to support multimodal interaction in order to provide interaction with complex, ontological data or even, more generally, for data browsing. For instance, Sonntag et al. [258] introduce a system called SmartWeb for entering a rich selection of Web-based information services in the context of semantic web technologies. Williamson, Murray-Smith and Hughes [297] describe an interface called Shoogle for multimodal data browsing on mobile devices. The reason for often using multimodality on mobile devices is the restricted availability of classical input and output options; often there is no physical keyboard and the display is very small and has a low resolution.

Another well-known application area for multimodal interaction techniques are virtual or augmented reality systems. Here, the main goal in developing interaction techniques is to approximate real life or to make interaction feel real like it has been described by Bowman et al. [31], as well as by Hand [105]. An example of gesture-based interaction in virtual reality environments is described by Latoschik [159]. A more general approach is described by Kaiser et al. [130], who use a 3D multimodal interaction for immersive augmented and virtual reality environments.

Multimodality is also the subject of investigation in formal interaction modeling. For instance, Navarre and his colleagues [180] introduced a dialog- modeling framework for modeling multimodal interaction and dialogs for virtual reality applications based on Petri net-based formalism. Another formalism based on Petri nets is described by Latoschik [160]. Katsurada, Nakamura, Yamada, and Nitta [132] introduce an XML-based approach for modeling multimodal systems called XISL.

Reeves et al. [230] propose that "multimodal interfaces should adapt to the need and abilities of different users, as well as different contexts of use. Dynamic adaptivity enables the interface to degrade gracefully by leveraging complementary and supplementary modalities according to

changes in task and context." These statements can be seen as further basis for this work in the adaption of user interfaces and as an indication of its possible extension to multimodality. Such extensions to the one user, one task, one interface scenario would take the first step towards more complex interaction scenarios.

Multimodality is often used as one conceptual part in the development of computer-supported learning systems [264]. Nevertheless, an important extension of computer-supported learning systems is support for cooperation or collaboration in learning (CSCL), like the CoBo system for learning cryptographic protocols based on cooperation developed by Baloian and the author of this dissertation [18, 283]. Cooperative and collaborative systems—not only in the context of learning systems—are also classical multi-user systems. Older studies by Patterson et al. [212] and by Benteley et al. [23] present architectures for computer supported cooperative work (CSCW).

More recent studies tend to focus on aspects of hardware components that offer multi-user interaction. For instance, Simon and Scholz [255] introduce an idea for projecting different image elements on a large-scale screen from multiple viewpoints to combine them in a single stereoscopic image. The goal is to enable multi-user interaction and collaboration in a virtual reality scenario. Cao, Forlines and Balakrishnan [37] took a one-user approach to handheld projector interaction and extended it to a multi-user scenario. The idea of handheld projector interaction is to extend the display and interaction space by projecting information on the physical environment. A common example of multi-user interaction supported by hardware based on the use of multi-touch tables is Microsoft's Surface®[3]. Another example of multi-user interaction using a multi-touch table is provided by Dohse, et al. [65]. They describe the use of a rear-projection, multi-touch tabletop enhanced by hand tracking. There are other examples using multi-touch tables as well, like those offered by Peltonen et al. [215] and Wigdor et al. [294], who both mainly describe empirical studies concerning the use of multi-touch hardware. One last study that should be mentioned is [63], in which Dini, Paternò and Santoro describe the use of handheld devices in a multi-user interaction scenario for improving museum visits through games. Its main goal is to enhance the learning experience by introducing game play to the museum visit.

Formal modeling and analysis approaches are not very widely spread and investigated for multi-user interaction and interfaces as is the case for single-user interaction. Only a few examples can be found from the 1990s onwards, and these were developed in the context of CSCW without any attempt to generalize them for general-purpose multi-user interaction. The study by Bentley et al. [23] described above is one example. Calvary, Coutaz and Nigay [35] present a generic software model for CSCW applications based on a modeling language called PAC*, which is based on approaches like MVC [99], PAC [50], or ALV [111]. This language describes mainly dialog models in a multi-user context using a graph-like visual representation that models the interaction between agents representing different components of the system. An example of analysis of multi-user interaction can be found in context of CSCL research. Mühlenbrock and Hoppe [175] describe a system that provides micro-analysis of the collaboration process.

As can be seen, formalization of multi-user interaction paired with validation, verification and analysis techniques has been the subject of rather less investigation. Still, many application scenarios and systems are multi-user scenarios and are sometimes more complex concerning such aspects as the responsibilities of the users or the cognitive state. This view of user interface

[3] http://www.microsoft.com/surface/en/us/default.aspx

modeling will be targeted in future work.

Upcoming chapters will introduce a formal approach to modeling user interfaces to define a general formal basis for investigating interaction and analysis to close the gap between user interface modeling and implementation.

3. Formal Modeling of User Interfaces

Formal modeling of user interfaces is one central aspect of this study. Nevertheless, every scientific study benefits from a solid nomenclature to withstand any possible misunderstandings and erroneous cross references to other definitions or identical terms in other scientific disciplines (Section 3.1). Based on this nomenclature, Section 3.2 describes a basic three-layered architecture for modeling of user interfaces. Next, Section 3.3 introduces a graph-based formal modeling language and its visual representation, which is especially useful for modeling the interaction logic of user interfaces. This formal model is then transformed into reference nets, a special type of Petri nets providing formal semantics for the graph-based modeling approach and an extensive set of tools for verification and validation (Section 3.4). Section 3.5 introduces various extensions to the formal modeling approach. Finally, Section 3.6 examines a group of XML-based description languages that can be used to model the physical representation of a user interface and specify its outward appearance. Here, a method for modeling interaction elements, such as buttons, will also be described; like the approach of modeling formal interaction logic, this method also uses formal description languages. The chapter will conclude with a summary of the concepts that have been introduced and give a short preview of the next chapter (Section 3.7).

3.1. Nomenclature

To avoid misunderstanding or miscomprehension, it is important to define the relevant nomenclature for the upcoming descriptions and specifications. It is not the aim of this section to give a universal nomenclature for all research areas and any possible application scenario in the context of HCI. The goal is to provide a basic nomenclature as background for this thesis dealing with the modeling of interfaces for the interaction of human users with technical machines and systems.

For example, the term *system* is of particular relevance to this study, but is an unclear term in science, defined in various ways in various fields. Here, it will be sufficient to examine several ideas of what the term *system* refers to without attempting to arrive at a universally valid definition. This will be also true for several other terms that have well established meanings in other research fields. Some additional terms instead are more or less new to the field of formal user interface modeling and will need to be newly defined.

System

The term *system* has been defined differently in various areas of research. For instance, Casti [42] introduces a definition of system from the perspective of engineering. Here, he is interested in defining the term according to system theory. He identifies five central questions or problem categories to be covered by system theory:

1. Realization-Identification

2. Reachability-Controllability
3. Observability-Reconstructibility
4. Stability-Bifurcation
5. Optimality of a system

The first category deals mainly with the problem of how to realize or characterize a system. Here, Casti characterizes a system as a "construction of a state-variable representation (internal model) for given input-output data (external model)." He continues by describing the underlying idea: "the state-variable model provides an explanation of the observed data, since the connections and data paths from the input to the output are explicitly exhibited." He concludes that "the state-variable model represents the internal workings of the black box called the system" [42, p. 10].

After identifying a way to describe the internal model of a system, for instance, using differential equations, the issue of what properties this model possesses in interaction with the outside world is of interest. One major property is the system's ability to reach other states from its initial state, where reachability involves "the initial state x_0, the terminal state x_T, the class of admissible inputs, and the time period during which the transfer $x_0 \rightarrow x_T$ is to take place" [42, p.12]. Closely related to the question of reachability is that of observability and reconstructibility. Casti states that "the intuitive content of the observability problem is that of identifying the precise amount of information about the system state that is contained in the measured system output" [42, p. 13]. Thus, observability of a system defines the extent to which the system's behavior can be reconstructed by taking its measured output data into account.

The last two major aspects of a system model—stability and optimality—mainly determine the qualitative aspects of a given model. The stability of a system model describes how the given model behaves under "perturbations of its defining structure" [42, p. 14]. The optimality of the model is measured by "superimpos[ing] an explicit criterion function upon the process and ... minimiz[ing] this performance index" [42, p. 17]. Thus, the optimality of a system is less oriented to an absolute perspective than its stability, but more oriented to a given performance measurement.

Still, Casti's engineering and math-oriented perspective on how a system is specified by a formally defined model accompanied by certain characteristics is too precise and goes too deeply into detail for the problem tackled in this work. It is not the aim here to model systems accurately but to try to model user interfaces to monitor and control such systems. Therefore, for the purposes of this study, the term *system* can be defined less specifically. All that is needed is an idea of what a system could be, how it can be modeled and how it is observed and controlled externally by a given user interface. Thus, the following statement describes a meaning of the term *system* that is closely related to the theory introduced by Luhmann [165]. There, he introduces the term *production* (among others) as one aspect of his definition of what a system is. According to Luhmann, one can refer to production if several but not all the reasons that are responsible for a certain outcome are under the control of a given system (translated from the German source [165, p. 40]). This term is closely related to the term *emergence*, which will be introduced in the following statement. Other examples of statements and definitions of the term *system* can be found in [58] and [108].

Statement 3.1 (System) *A* (closed) system *is a well-defined part of the universe. A system consists of elements that interact in a more or less complex way and are in relationship with each other. Any element and any specific interaction and relationship is specified by its individual characteristic.*

A system can show characteristics that are neither characteristics of elements nor of interactions and relationships. Such characteristics are called system characteristics *or* emergences*.*

A system is called dynamic *if the characteristics of elements, interactions and relationships change over time.*

The system state *of a given system is the collection of all characteristics (including emergences).* ◇

Statement 3.1 touches on various aspects familiar from system theory without trying to be a universal definition of *system*. It has a clear technical focus relevant for this work. First of all, the term *element* is used to describe things in the universe that cannot be further split (in a technical or physical sense). Thus, these things called *elements* can be characterized as *atomic*. A second term in the above statement is that of *emergent characteristics*, here also called *system characteristics*. The emergent characteristics of a system are observable on a macroscopic level. These characteristics cannot be matched directly to the systems' elements or to their interactions and relations. These characteristics become visible exclusively through observation of the whole system.

For instance, a water molecule is a system in the sense that it is constructed of two hydrogen atoms, one oxygen atom and their electro-physical interactions. The boiling point of water is more or less $370°K$, depending on barometric pressure. Pressure can be interpreted as interaction with the (surrounding) universe of the system *water molecule*. This interaction with the environment is not covered by statement 3.1, which only refers to closed systems that do not explicitly interact with their environment. Still, the characteristic of the system *water molecule* called *boiling temperature* is emergent. It cannot be deduced from the boiling point of hydrogen, which is $20.28°K$, or from that of oxygen, which is $90.2°K$. Moreover, the electro-physical binding between hydrogen and oxygen does not have a boiling temperature in this sense. Only through the concurrence of all elements, interactions and relationships can one determine the boiling temperature of $370°K$. The extent to which emergences result from the high complexity of interrelationships between the elements of a system might also be discussed, but that is unnecessary in this context.

A further important aspect of systems to be considered is their connection to and interaction with the surrounding universe, also denoted as the environment of a system. Therefore, the terms *open* and *closed system* can be defined as differentiated systems that interact with their environment (open systems) and differentiated systems that do not interact with it (closed systems). Still, defining the term *open system* in this way is different to system theory approaches in engineering such as the one offered by Casti [42]. In this context, open systems are generally unstructured, in contrast with the formally structured nature of systems as defined by Casti. Thus, open systems in engineering terminology can be seen more in the role of describing the environment. In this work, open systems must be defined differently to stay consistent with the term system as introduced above (Statement 3.1) and to be able to define a system interface as a specific set of observable and manipulable system characteristics. This definition is needed

for the further development of the formal modeling approach of user interfaces as the central research goal of this work, without explicitly excluding any invisible information exchange between the system and its environment besides the formally defined interaction through the user interface.

> **Statement 3.2 (Open System)** *An* open system *interacts with its environment. The environment is able to observe and manipulate the characteristics of the system's elements, as well as the elements' relationships and interactions. Observable and manipulable characteristics of a system are characteristics that are visible to an observer who is not part of the system.* ◇

Again, statement 3.2 is very general. It is helpful to define the term more specifically by summarizing the system's ability to interact with its environment. As noted above, the ability to interact with the environment makes it an open system. Therefore, the following definition of the term *system interface* refers to a well-defined set of system characteristics and relationships that are observable and/or manipulable. Representing a system in the context of an environment in this way simplifies the formal modeling of user interfaces in the upcoming sections.

> **Definition 3.3 (System Interface)** *A* system interface *is a well defined and finite set of characteristics of the elements of a system and characteristics of the elements' relationships and interactions that are observable and manipulable by the environment. System characteristics or emergences can only be part of the system interface if they are exclusively declared as observable and not manipulable.* ◇

Definition 3.3 seeks to define a compromise between closed systems, which do not interact with their environments but can be observed from outside, and open systems, which interact with their environments in unspecific and unpredictable ways. Thus, a *system interface* makes it possible to define systems that interrelate with their environments in a well-defined way. Especially in the context of technical systems, this definition seems a suitable approach for further descriptions and makes a formalization of interaction possible. Emergences are not manipulable because they result from the elements and their interactions. Thus, emergences are only manipulable indirectly through the manipulation of involved system elements and their relationships.

Figure 3.1 compares open and closed systems in the context of the system interface definition given above (Definition 3.3). Therefore, a system S_1 is represented as set of characteristics (emergent or non-emergent/observable/manipulable or unobservable) shown as a thick and vertical line or block in Figure 3.1. Those characteristics which are observable or manipulable are part of the "system block" located above the central axis of the diagram in Figure 3.1. The block above the central axis is further divided into two parts, one for the manipulable characteristics of the system and one for system characteristics that are only observable. Invisible characteristics are part of the block below the central axis; thus, they are neither observable nor manipulable from the environment of the system. The system interface is shown as a white block in the system block and is interpreted as a well-defined subset of observable and manipulable characteristics (as defined in Definition 3.3). To differentiate open from closed systems, a dashed vertical line in the middle of the diagram separates the two. S_2 is an example of a closed systems, where the environment is not able to manipulate any characteristics but can still observe several. A subset of these observable characteristics are part of S_2's system interface

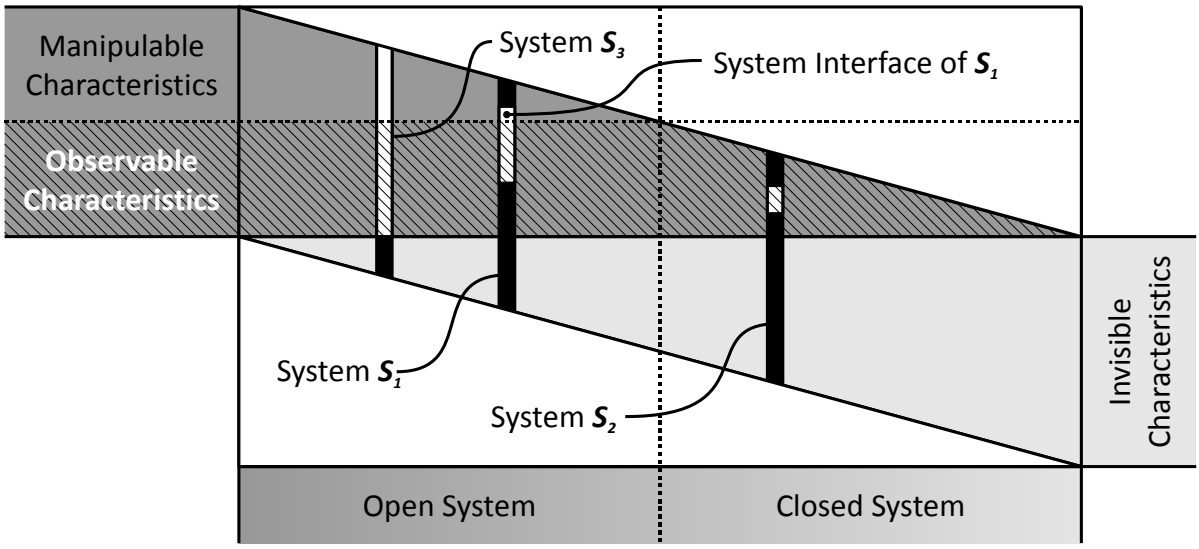


Figure 3.1.: Comparison between open and closed systems showing the unspecified interaction of the system with its environment on one side (open system) and the invisible characteristics of the system on the other (closed system)

and can thus be presented as output on a user interface. Manipulable (and observable) characteristics are not part of the system interface but interact with the environment in a unknown fashion. It is not necessary to define precisely which other characteristics of a system are observable and how these are influenced by the environment. In fact, in the context of this work, only characteristics that are part of the system interface are considered in the user interface model: the internal structure of the system and other observable or invisible characteristics are ignored. Still, the definition is flexible enough to introduce such aspects into the focus of future related work if necessary in order to, for instance, generate more finely granulated user interface models.

Figure 3.1 further shows a specific example of a system denoted as system S_3. S_3's observable characteristics are all part of its system interface. Thus, this kind of system is completely "controllable" in the sense that all its observable or manipulable characteristics can also be addressed in a well-defined way through its system interface. This also means that there is no undefined or unobserved interaction between the system and its environment. Still, in real scenarios, such as one in which a technical machine acts as a system, there will always be unobserved interaction with the environment. Therefore, example system S_1 represents the scenario generally relevant to the modeling of user interfaces.

Any scientific field uses models to describe complex systems, interrelationships or, in general, *things*. In this way, science tries to reduce complexity, using models to handle and understand systems, exclude the complex environment and deduce a well-defined interaction between the two. Because of this, the term *model* is used in various scientific disciplines with differing meanings. This makes a clear definition of the term necessary.

Definition 3.4 (Model) *A* model *is an artifact of scientific methodology that describes the investigated reality using parameters and their relationships in the context of a scientific theory using a specific formalism. Observable parameters are connected by causal laws.* ◇

Now it is possible to bring Statement 3.1 of the term *system* and Definition 3.4 of the term *model* together, resulting in a term describing a model-based representation of a system—a *system model*.

> **Definition 3.5 (System Model)** *A* system model *is a model description of a system that describes its elements identified by their characteristics as well as their interrelationships and interactions using formal, semi-formal or informal description methods.* ◇

A *system model* is a model of a system describing its elements and their interrelationships in a more or less formal way. Normally, a system model simplifies its real correspondent to make it expressible and understandable (in a formal way). For instance, in physics a model of a spring is a differential equation describing its behavior on a macroscopic level without taking the behavior of all atoms and molecules of the spring's material into account. This example also shows how system models can be simplified to represent adaptions. The differential equation describing a spring can take damping into account as an interaction with the environment; alternatively, it can delete this factor from the model, resulting in an ideal spring. Even though an ideal spring does not exist, the model helps one understand how a spring works.

In the context of this dissertation, a special type of system model is of interest. Talking about modeling user interfaces can address interfaces for the interaction of a human user with a data processing system or, more specific spoken, with a computer system.

> **Definition 3.6 (Computer System Model)** *A* computer system model *is a system model that describes a data processing system using a formal, semi-formal or informal description method. This type of model is also called* action logic. ◇

Wherever the term *system* is used below, it is generally data processing computer systems that are meant. They are the main focus of HCI studies—in addition, of course, to the human who forms the other half of the interaction.

The next section focuses on the interaction of *users* with complex systems. Therefore, the term *user* will be defined on the basis of a nomenclature arising from interaction with systems.

Interaction

Based on the statements and definitions introduced above, the following explanations and definitions will define interaction between a system and its surrounding environment.

> **Definition 3.7 (Interaction)** *Data flow between a system (interface) and its environment is called* interaction. *This data exchange is differentiated between input and output. Data flow emitted by the environment that influences the system is called input; data flow from the system to the environment making system states visible to the outside world is called output.* ◇

Applying this definition to the term *system interface* means that the system interface is a set of input and output flows of data that are well defined through the system interface itself. Based on a given system interface, interaction can be formalized as a set of data exchange processes using a suitable formalism. Normally, interaction is much more complex than input and output data. The way this data (in a certain data format) is generated or processed is important in

the context of HCI, as will be discussed below. Thus, a formal description of interaction as a set of data flow processes must be defined and will be summarized in the following definition as *interaction logic*. Here, *logic* can be understood in the mathematical sense, where it formalizes the mutual dependencies of elements (or variables).

> **Definition 3.8 (Interaction Logic)** Interaction logic *is a model of interaction representing a set of models of* interaction processes. *Interaction processes model processes of data flow between a system interface and its environment. Interaction logic can be described formally, semi-formally or informally.* ◇

Definition 3.8 defines *interaction logic* as a set of interaction processes that model how data emitted by the system interface is processed to be presented to the system's environment. Vice versa, it describes how data originating in the environment is processed to be sent to the system interface. This is only true if there has been no more interaction than that defined by the interaction logic. In general, no more data flow can be modeled by interaction logic than is predefined by the system's interface. This restriction results from the above definitions and comments on open systems and system interfaces. It allows further considerations to be neglected due to proofs of completeness of system interfaces or interaction logic. Still, the meanings of the terms *system* and *interaction* introduced above would not allow such proofs because of the insularity of *system interfaces* (cf. Definition 3.3).

In HCI research, it is also necessary to define the quality of interaction logic. In this regard, the terms *adequate* and *appropriate* interaction logic will be defined to take a look at human error in interaction and why it occurs in bad interaction logic.

> **Statement 3.9 (Adequate Interaction Logic)** *Interaction logic is called* adequate *if it supports all kinds of interaction processes such that all possible system states of a system can be changed through interaction to a final system state. A* final system state *is a system state that is final with regard to a given task to be solved using the given system.* ◇

Thus, a final system state is associated with a task that a user tries to complete using the system. Still, this definition of interaction logic does not restrict interaction such that it avoids critical system states. For this reason, Definition 3.10 identifies further restrictions.

> **Statement 3.10 (Appropriate Interaction Logic)** *Interaction logic is called* appropriate *if it is adequate and, furthermore, does not support interaction processes reaching unwanted system states. A system state is* unwanted *if it shows characteristics of the system that can be called abnormal or that are not part of the system's specifications.* ◇

A strategy for creating an appropriate interaction logic is to configure a user interface that makes it impossible for the user to change the system state to an abnormal one. Another option is to identify relevant interaction processes in a given interaction logic and reconfigure the interaction logic to prevent the occurrence of abnormal system states. This type of *reconfiguration* can have various triggers. The reconfiguration can be triggered by an observer of the system or by the user. These aspects discussed in Chapter 4, where a formal definition and the formalism for reconfiguring user interfaces will be introduced. This kind of reconfiguration does not change the action logic, nevertheless, it is a complete or incomplete model of the real

system. It is still possible for the system state to become abnormal, but this should be inhibited by the interaction logic.

This is a good point at which to discuss to what extent it is possible to prove the appropriateness, or adequacy, of an interaction logic and to what extent an interaction logic or, more generally, the design of a user interface can avoid the occurrence of unwanted system states and—probably more importantly—how interaction logic can offer interaction processes for bringing a system back to a normal system state after a critical one has been reached. This is an aspect of formal verification that is part of future work.

Up to now, interaction has been defined and classified from the perspective of the system but not from that of the environment. For the purposes of this discussion, the environment can be reduced to another system, without specifying its system interface and its interaction logic in a concrete way. In the following, this system will be called *user*, referring to a human user who interacts with a data-processing (or other) system. Thus, a user can be seen as a system in that a user can receive data and react by sending data in response. This view of a human user corresponds to the view of the human mind in cognitive psychology: Cognitive science seeks to identify cognitive modules (corresponding to system elements) and their relationships (corresponding to relationships between system elements). For an example, see the work of Anderson [5]. A further reason to liken a human user to a system is an intended dependence of systems to its interaction, modeled as interaction logic. To close the loop, a mental model (cf. Section 2.2) is nothing other than a model of a *human system* that makes use of the analogy of a human to a system.

The following statement helps to classify different specifications of the term *user* that will be used in subsequent sections and chapters.

Statement 3.11 (User) *A* user *is a system interacting with another system. The following types of users can be specified:*

- Human User*: A human using a (data-processing/computer) system to solve a given task, referred to below by the single term user.*
- Expert*: A special human user who has a complete and correct mental model of a (data-processing/computer) system. From this user's interaction, an expert model can be derived demonstrating the error-free use of the system to reach a certain goal.*
- Learner*: A special human user who is learning a given task or content through practice and training. A learner does not necessarily learn to interact with a given system.* ◇

The last step in this section will be to define the term *user interface*, which is closely connected to the term *interaction element*.

Statement 3.12 (User Interface, Interaction Element) *The task of a* user interface *is to influence specific characteristics of a system and its system state and to present specific, observable data of that system to the outside environment.*

A user interface is composed of a set of interaction elements *that are defined by their physical and logical characteristics. The physical characteristics of an interaction element describe its outward appearance; they include position, size and coloring. The logic characteristics of an interaction element describe the type of interaction*

involved: (a) what kind of data is generated, (b) how data is presented to the environment and (c) how input and output of data is combined (called hybrid or complex interaction elements). The entire set of the interaction elements of a user interface comprises the physical representation *of the given user interface. The flow of data and the interaction with a system using a user interface is defined by a certain* interaction logic *associated with the physical representation and the system interface of the system to be controlled.* ◇

In conclusion, a user interface can be modeled as a three-layered architecture: (a) the physical representation describing the outward appearance as set of interaction elements, (b) the system to be controlled represented in the form of its system interface and (c) the interaction logic that models the flow of data between the two.

3.2. User Interface Modeling

The architecture presented in this section is based primarily on mental models as cognitive representations of complex systems as described in Section 2.2 and as results of the nomenclature introduced above. Modeling the cognitive representation of a machine with mental models means representing the functionality of a machine with a set of cues that are activated by perceived external data (e.g., data presented by the user interface). This enables the user to interpret the current system state, consider new steps in the interaction process and predict the future system state. These data and the reaction of the user perceiving them are directly influenced by the user interface and its behavior. The behavior of a user interface can be defined as how the system state or parts of it are presented to the user and how input data that is generated by the user is processed and sent to the system. By creating a language for modeling the behavior of a user interface, it becomes possible to introduce a specific, cue-based mental model into the user interface without the need to integrate different and incompatible formalisms and modeling approaches. The result will be a reduction of the gap between the user's mental model and the behavior of a user interface and, as a result, a reduction in interaction errors. This hypothesis will be investigated in Chapter 7, showing that reconfiguration and adaption of interaction logic influences the amount of human error in interaction.

Formal modeling needs to enable modeling of the behavior of a user interface. Therefore, the architecture in Figure 3.2 distinguishes two layers of a user interface and the corresponding layers representing the system interface (as described above): (a) the physical representation of the user interface and (b) its interaction logic. The physical representation of a user interface shows which interaction elements are positioned at which part of the interface and which visual parameters are set to them (color, size, shape, etc.).

Interaction logic instead describes the behavior of a user interface previously defined as the data transfer to and from the system to be controlled. The behavior of a user interface can be understood as a type of data processing of input and output data to and from the system to be controlled by the user interface. Thus, interaction logic in our sense is a set of interaction processes each describing the processing of events resulting from interaction elements on the physical representation layer or from the system interface, for example, a change of system state. Interaction processes themselves can also interact with one other, resulting in complex interaction logic behavior. In the context of cognitive psychology (here, mental models), an interaction process that is activated by such means as pressing a button should process the

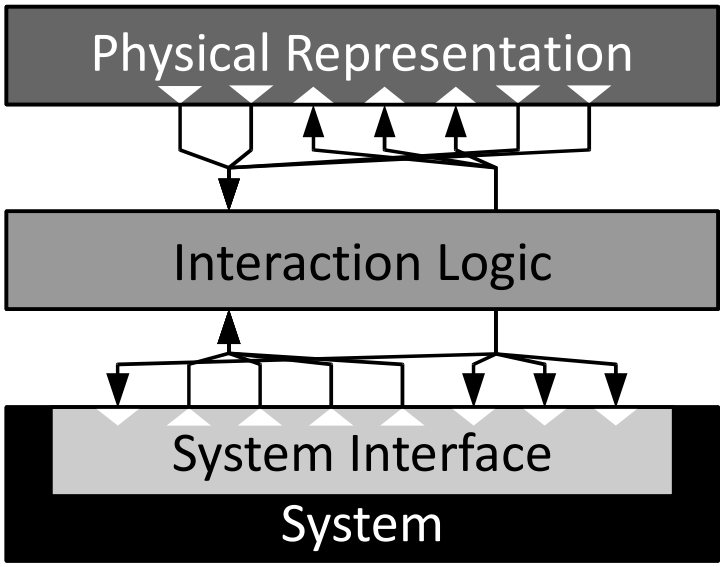


Figure 3.2.: Three-layered architecture for formal modeling of user interfaces: An approach based in cognitive research and the terminology of mental models.

resulting input data exactly according to the user's expectation (here the cue of the mental model). The user's expectation is influenced by the physical representation of the user interface but also by the user's experiences and expectations as to how the entire system will work; together, these form the user's cognitive model. Using interaction logic as a second layer of a user interface model, a formal mental model (as a set of cues) can be directly transformed into a user interface. Vice versa, formally defined interaction logic can be matched algorithmically to a given mental model as part of formal verification analysis.

The main goal of this study is to maintain a formal basis as a platform for transferring results from HCI and psychology to the creation and adaption of user interfaces. Therefore, a good starting point is to formalize the (informal) architecture of a user interface introduced above. The following definition provides an initial formal structure: a formal user interface.

Definition 3.13 (Formal User Interface) *A* user interface UI *is a three tuple* $UI = (PR, IL, SI)$*, where* PR *is the physical representation, and* IL *is the interaction logic of a user interface.* SI *is a given system interface.* ◇

The precision of the interaction logic (IL) will be investigated below. The physical representation PR can further be formalized as follows.

Definition 3.14 (Physical Representation) *A* physical representation PR *is a tuple* $PR = (I, P, C, p, c)$*, where* I *is a finite set of interaction elements,* P *is a finite set of physical parameters and* C *is a finite set of classes or types of interaction elements.* p *is a total function where* $p : I \to P^*$ *matches any interaction element in* I *to a set of physical parameters* P'*, where* P^* *is the set of all subsets of* P*, such that* $P' \subseteq P$*.* c *is a total function for the classification of interaction elements, where* $c : I \to C$ *matches any interaction element in* I *to a class in* C*.* ◇

Definition 3.14 defines a physical representation of a user interface as a set of interaction elements where every element is associated with a set of physical parameters $p_i \in P$, defining such

characteristics as their outward appearance. Furthermore, interaction elements are classified by a classification function c, which associates any interaction element $i_j \in I$ with a class $c_k \in C$. Three classes of interaction elements are of most interest: (a) interaction elements for input, (b) interaction elements for output data and (c) complex interaction elements that combine input and output data. This basic classification can easily be extended by more specific classes such as *data visualization* or a class of *editing* interaction elements. A closer look at the formalization and representation of the physical representation will be introduced in Section 3.6 mainly on the basis of XML-based description languages and a new modeling approach using formal graph-based models.

The system interface was introduced informally above as a term. Here, to be consistent with the formal Definition 3.14 of *physical representation* and to offer a starting point for formal interaction logic, which will be discussed below, a more formal specification will be given.

> **Definition 3.15 (System Interface)** *A* system interface $SI = (SI_I, SI_O)$ *is a pair of sets, where* SI_I *is a set of input variables and* SI_O *is a set of output variables, where* SI_I *and* SI_O *are disjoint. All variables of sets* SI_I *and* SI_O *are associated with a data type. Input variables of set* SI_I *are write-only variables, whereas variables of set* SI_O *are read-only variables.* ◇

Definition 3.15 differentiates strictly between input and output variables and thus between values that trigger changes of the system state (values entered into the system by being bound to input variables) and values that represent parts of the system state (values passed to the environment through output variables). In this way, input variables are variables that can be sent to the system from outside but cannot be read. In such cases, a change in the system state can result in changes to input values through their relationship to other (also invisible) elements of the system. Output variables are variables that can be read, but not written. They represent system values that can be observed representing the system state or parts of it to the outside environment. Values that are set to variables of set SI_I can be used beforehand for other data processing purposes and modeled as interaction process in interaction logic. Still, one variable can be part of sets SI_I and SI_O, resulting in a variable that can be both written and read. Nevertheless, depending on the context, a given variable's role is well defined.

The next section discusses interaction logic as the third part of the formalization of user interfaces in detail.

3.3. Formalization of Interaction Logic

Interaction logic is defined above as a set of interaction processes. Another aspect of interaction logic is the dependencies and relations between two or more different interaction processes. Modeling these kinds of interaction processes, a language has to be presented to an (expert) user such that it offers a visual way of modeling interaction logic paired with a formally defined background for such purposes as transforming a model into a executable, validated, and verifiable formalism in order to verify interaction logic against a given action logic or an expert model of a given interaction logic. For instance, state charts [106] or activity diagrams [195] as part of the UML modeling language standard are examples for modeling processes using visual languages. In economics, languages like Business Process Modeling Notation (BPMN) [197, 290] as visual representations of Business Process Execution Language (BPEL) [191] are

of great use in modeling business processes in production or document management. Still, these languages are often very general and therefore are not specifically suited for the concrete problem of modeling interaction logic. The following sections introduce a formalism that is tailored to the area of modeling interaction logic but still uses formalism introduced in BPMN to extend the set of possible implementation of interaction processes. It is also easily transformable to reference nets [149], a special type of Petri nets. The following two subsections describe the modeling language FILL for the formal modeling of interaction logic as a graph-based structure and its visual representation, called VFILL. Both were first introduced in [286].

Formal Interaction Logic Language

This section begins by introducing a graph-based formalism called FILL for describing data flow in interaction logic. It goes on to explore how specific aspects of BPMN and a Java-like type system are used to extend this basic approach.

Definition 3.16 (Formal Interaction Logic Language - FILL) *The* Formal Interaction Logic Language *(FILL) is a 9-tuple*

$$(S, I, P_I, P_O, X_I, X_O, P, E, \omega),$$

where S is a finite set of system operations, and I is 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, such that S, I, P_I, P_O, X_I and X_O are pairwise disjoint. P is a finite set of pairs

$$P = \{(p, o) \mid p_I(p) = o\} \cup \{(p, o) \mid p_O(p) = o\},$$

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

$$\begin{aligned} \forall s \in S \quad &: \quad (\exists_1 (p, s) \in P : p_I(p) = s) \wedge (\exists_1 (p, s) \in P : p_O(p) = s), \text{ and} \\ \forall i \in I \quad &: \quad \exists_1 (p, i) \in P : p_O(p) = i. \end{aligned}$$

E is a finite set of pairs, with

$$E = \{(p_O, p_I) \mid e(p_O) = p_I\},$$

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

$$\forall (p_O, p_I) \in E : (p_O \in X_O \Rightarrow p_I \notin X_I) \wedge (p_I \in X_I \Rightarrow p_O \notin X_O). \qquad \diamond$$

In Definition 3.16, there are two types of operations: system operations (as elements of set S) and interaction-logic operations (as elements of set I). System operations describe connections to and from the system, thereby representing the system interface of a system in interaction processes of certain interaction logic. Concerning Definition 3.15, system operation is differentiated into system operation for input data to the system and output operation to read out system values. This is consistent with Definition 3.15 and the nomenclature Definition 3.3. Thus, passing a value to a system operation influences the state of the system. After completing this change, the passed value is returned and thrown back into the interaction process. The same is true of system operations that return a value from the system, such as showing a special part of the system state. Here, a data value passed to the (output) system operation will be

consumed, and triggers the associated system value to be outputted at the output port of the system operation back into the interaction process. This returning functionality of passed data to system operations makes it possible to trigger downstream nodes in an interaction process, without changing (input system operations) or consuming and passing a new data object (output system operation) to the interaction process. Here, one important ability of FILL becomes apparent: Processes are modeled as data passing from one node to another in a graph-based model. This makes a FILL model executable and transformable to a colored Petri net-based formalism, which models data passing as token creation and consumption.

Interaction-logic operations process data passed to them in various ways, including data type conversion and mathematical calculation. This data processing can be implemented in a programming language like Java. In general, interaction-logic operations represent data processing methods that may be more or less complex. Later on, it will be shown how data processing methods can be directly introduced into an underlying reference net derived from a FILL graph.

Any operation is associated with several input and/or output ports (as elements of sets P_I and P_O), where this association is defined by function p. These ports are used to connect an operation to another operation or to a proxy. Proxies are elements of sets X_I and X_O, which connect an interaction process to physical interaction elements on the physical layer of a user interface. Thus, output proxies represent interaction elements, like buttons, for input data to the interaction process, and input proxies represent interaction elements for outputting or presenting data resulting from an interaction process to the user, like labels. The edges of a FILL graph are directed edges between proxies and ports. Every edge in a FILL graph points from an output port or proxy to an input port or proxy. Also, any port or proxy is attached to exactly one or no edge. These constraints make a later conversion to reference nets much easier without constraining expressiveness and while taking the extensions discussed below into consideration. Nevertheless, the consumer node, or terminator, ω is a special node that identifies the end of an interaction process and consumes all data objects that are sent to this node.

In the extended version of FILL, a third type of element is introduced that offers various ways of branching and combining interaction processes, such that more complex interaction processes can be modeled and the above-introduced 1:1 edge restriction does not restrict the expressiveness of FILL. These elements are taken from BPMN [197, 290], a visual language for modeling business processes. A second part of this extension is a new type of operation, called a *channel operation*. These operations enable interaction between various interaction processes.

Definition 3.17 (Extended Formal Interaction Logic Language - FILL$_{EXT}$)
FILL$_{EXT}$ is a 17-tuple

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

where S, I, P_I, P_O, X_I, X_O, and ω are identically defined like for FILL. 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

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

S, I, C_I, C_O, P_I, P_O, X_I, X_O and B are pairwise disjoint.
P' is a finite set of pairs extending P from basic FILL, such that

$$P' = P \cup \{(p, o) | p'_I(p) = o\} \cup \{(p, o) | p'_O(p) = o\},$$

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

$$\begin{aligned} \forall c \in C_I \quad &: \quad (\exists_1 (p,c) \in P' : p'_I(p) = c) \wedge (\nexists (p,c) \in P' : p'_O(p) = c), \text{ and} \\ \forall c \in C_O \quad &: \quad (\exists_1 (p,c) \in P' : p'_O(p) = c) \wedge (\nexists (p,c) \in P' : p'_I(p) = c). \end{aligned}$$

E' *is a finite set of pairs extending* E *from basic FILL, such that*

$$E' = E \cup \{(p,b) \mid e'(p) = b,\ b \in B\} \cup \{(b,p) \mid e'(b) = p,\ b \in B\},$$

where $e' : P_O \cup X_O \cup B \to P_I \cup X_I \cup B \cup \{\omega\}$ *is a function, extending* e *from basic FILL, and*

$$\begin{aligned} \forall b \in B \quad &: \quad (\#\{(p,b) \mid (p,b) \in E'\} > 1 \Rightarrow \exists_1 (b,p) \in E') \ \vee \\ &\qquad (\#\{(b,p) \mid (b,p) \in E'\} > 1 \Rightarrow \exists_1 (p,b) \in E'). \end{aligned}$$

l *is a function with*

$$l : E' \to \mathcal{L},$$

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

$$g : B \to \mathcal{B},$$

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

$$c : C_I \to C_O. \qquad \diamond$$

In Definition 3.17, there are three extensions to FILL's basic definition (Definition 3.16). The first is the introduction of nodes from BPMN [197, 290]. The main goal of this extension is to introduce a node type for fusing and branching various interaction processes in interaction logic. This fusion and branching offers a conditioned way to control the data flow without applying complex data-processing operations that is subject to an interaction-logic operation. For instance, in the case of processing data resulting from a multimodal interaction like speech combined with tactical input, the combination of this data should be carried out by an interaction-logic operation offering two input ports and complex processing functionality. The result would be one output value. At the same time, every interaction-logic operation also has semantics of fusing data objects to one output data object.

A closer look into the semantics of the three BPMN nodes in an interaction process is presented in Table 3.1. In general, these nodes have either 1 incoming and n outgoing or n incoming and 1 outgoing edge. The former (1 to n) is called a *branching node*; the latter (n to 1) is called a *fusion node*. This has been formalized in Definition 3.17 by the restriction to E' concerning all BPMN nodes $b_i \in B$. The main reason for excluding the n to n structure is to simplify the semantics of BPMN nodes, which would become much more complex if n to n is considered a valid structure in FILL_{EXT}.

Thus, for any BPMN node in FILL_{EXT}, fusion and branching have to be separately described and specified. BPMN nodes are activated after one $m < n$ or n incoming data objects (depending on the type of node and if it is a branching or a fusion node) are delivered to the node.

Table 3.1.: Informal specification of BPMN node's semantics in FILL$_{EXT}$

Node Type	Branching	Fusion
AND	All outgoing edges are activated at the same time, in parallel if the incoming edge is activated ⇒ Parallelization. The incoming data object will be transmitted as copy to all outgoing edges.	All incoming edges must be activated before activation of the outgoing edge ⇒ Synchronization. Here, it is necessary to define which data object of which incoming edge will be transferred to the outgoing edge using a guard condition.
OR	Depending on the node's guard conditions, one outgoing edge or a group of outgoing edges will be activated upon the activation of the incoming edge.	Depending on the nodes's guard conditions, one edge or a group of edges will be synchronized.
XOR	Depending on the node's guard condition, exactly one outgoing edge will be activated if the incoming edge has been activated by an incoming data object.	Depending on the node's guard condition, exactly one incoming edge must be activated before a specified outgoing edge is activated by the incoming data object.

The decision as to which outgoing edge will be activated, or even whether any edge will be activated, depends on the type of data and its value as well as on the conditions given as Boolean expressions, the so-called *guard conditions* of the BPMN node. For the detailed semantics of BPMN nodes, see Table 3.1.

An introduction of BPMN nodes as related to FILL would be incomplete without defining two terms that are also valid in the context of general operation nodes in FILL$_{EXT}$ graphs. Furthermore, the following descriptions introduce informal semantics for FILL$_{EXT}$ graphs.

> **Edge activation**: An edge is activated if a data object is send to or from a node via that edge.
>
> **Node activation**: A node is activated if edges have been activated as defined in Table 3.1.

In the case of operation nodes, an outgoing edge is activated after the calculation (that was triggered by activating this node) has been completed or the value from the system has been returned. Edge activation of outgoing edges in BPMN node activation can be additionally specified by guard conditions that are associated with the BPMN node. Therefore, an evaluation sequence for BPMN nodes is necessary and can be described as follows.

1. **Evaluation of edge activation of incoming edges** The activation of incoming edges triggers the activation of a BPMN node for fusion as follows: an XOR node is activated if one incoming edge is activated; an OR node is activated if a specific group of incoming edges is activated; an AND node is activated if all incoming edges are activated. In the case of branching, a BPMN node is activated if its incoming edge is activated.
2. **Evaluation of the BPMN node's guard condition(s)** that define(s) what data is passed to the outgoing edge (fusion case), which outgoing edge(s) is (are) activated

(branching case), and whether the outgoing edge(s) will be activated or not.

Considering the need for Boolean expressions, a further extension has been introduced. In FILL$_{EXT}$, BPMN nodes can be associated (defined by function g) with Boolean expressions as elements of set $\mathcal{B}$ that are evaluated on the data passed through the edges to the BPMN node. This data is identified by labels that are attached to incoming edges and thus can be used as variables in the guard conditions. For instance, in the case of fusion, a guard condition has to specify which data object from which incoming edge should be passed to the outgoing edge under which condition. This makes it necessary to label outgoing edges, too, so that they can also be referenced in guard conditions. How the expressions are to be formulated or modeled depends on the formalism used for further processing of FILL models. In this dissertation, reference nets are used. This Petri net formalism implements a special object-based language for edge, transition and place inscriptions that is closely related to JAVA programming language and its syntax for propositional logic expressions. In reference nets, conditions that control the activation and firing of transitions are called *guards*, which are closely related to FILL's guard conditions.

The third extension of FILL is the introduction of channel operations, that model communication between different interaction processes. The referencing mechanism in reference nets can be used to represent channels (in FILL$_{EXT}$ represented by function c) between channel operations. These channels between an input- and an output-channel operation pass the data sent to an input-channel operation to all output-channel operations associated with that input-channel operation via data duplication. It should be mentioned that a channel is a pair (c_I, c_O) composed of an input-channel operation c_I and an output-channel operation c_O with $c(c_I) = c_O$ of a given FILL$_{EXT}$ graph. Thus, a channel always has exactly one input and one output-channel operation associated to it. Still, input and output-channel operations can be associated with n different channels as indicated by the definition of c as a relation.

A further extension of FILL$_{EXT}$ is the introduction of an object-oriented definition of data types. This extension provides many benefits, for instance, type safety for data objects passed through the FILL$_{EXT}$ graph and simpler transformation to reference nets, whose tokens are also associated with Java-like object-oriented data types. This extension makes it possible to model data flow in a more expressive way. Thus, on a semantic level, FILL$_{EXT}$ not only describes how data is processed in an interaction process, but it also describes what kind of data is passed from one operation to another.

Definition 3.18 (Typed Formal Interaction Logic Language Extended - TFILL$_{EXT}$) *TFILL$_{EXT}$ 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, I, C_I, C_O, P_I, P_O, X_I, X_O, B, P', E', l, g, c, ω, $\mathcal{L}$, and $\mathcal{B}$ are identically defined like for FILL$_{EXT}$.
T is a finite set of data types and t is a total function with

$$t : (P_I \cup P_O \cup X_I \cup X_O) \to T. \qquad \diamond$$

In Definition 3.18, FILL$_{EXT}$ is extended to TFILL$_{EXT}$ by using a set of data types T and a function t that assigns a data type to every port and proxy. On the semantic level, this means that a port of a given type only accepts data of that type. Thus, for example, a port of type

String only accepts data of type String or its sub-types. The structure of elements of set T is not more clearly specified so that TFILL$_{EXT}$ will be open to different type systems like that of Java or C++. This structure reflects how different types are organized and how the semantics is defined. For instance, if the type system offers inheritance, types can be super-types of other types. This has to be separately specified for the elements of set T. Due to the use of reference nets as a Java-based simulator, the type system implemented for Java will be used here.

In conclusion, TFILL$_{EXT}$ has the following features:

1. Graph-based modeling of interaction processes that describe the interaction logic of a user interface. Here, the connection to the physical representation of a user interface is represented as special types of nodes in the interaction process called proxy nodes.

2. Three different types of operations were defined: (a) system operations, which represent values to be assigned to or read from the system (defined by the SI of the system) to be controlled by the user interface, (b) interaction-logic operations that offer more complex data conversion and processing in the interaction logic with high flexibility due to the integration of high-level programming languages and (c) channel-operations that connect different interaction processes in the interaction logic with one another.

3. BPMN nodes are introduced to extend the ability to model complex processes based on branching and fusion.

4. Edges can have labels, and BPMN nodes can be associated with Boolean expressions referring to edges' labels. BPMN nodes and conditions associated with BPMN nodes offer a more complex modeling of interaction processes.

5. Nevertheless, TFILL$_{EXT}$ is type safe concerning data that is passed through the model. All connector elements (ports and proxies) are associated with object-oriented data types.

In the following sections, the abbreviation FILL will be used to include TFILL$_{EXT}$ as well. FILL is designed to be easily transformed to reference nets, which provides a formal semantics for FILL, and also provides a theory of graph transformation [74], which can be used to define formal user interface reconfiguration [284]. Also, verification and validation methods widely investigated for Petri nets can be used in our context in various ways, such as to identify failures in a user interface and to describe formal usability. These topics are the subject of future research.

FILL is basically not exclusively motivated as a process-modeling language for modeling interaction processes. Closely related to interaction process modeling are modeling approaches for business processes. FILL has been also developed for interactive modeling of interaction logic supported by visual editors like the one described in Chapter 5. Therefore, a visual representation called Visual FILL or VFILL was developed to make it possible to model FILL graphs visually.

Visual Formal Interaction Logic Language - VFILL

The previous section defines FILL on a formal, syntactic, and graph-based level. Nevertheless, FILL was planned to be a visual language for modeling by the use of interactive tools. Thus, this section presents the visual representation of FILL, called VFILL, which stands for Visual FILL.

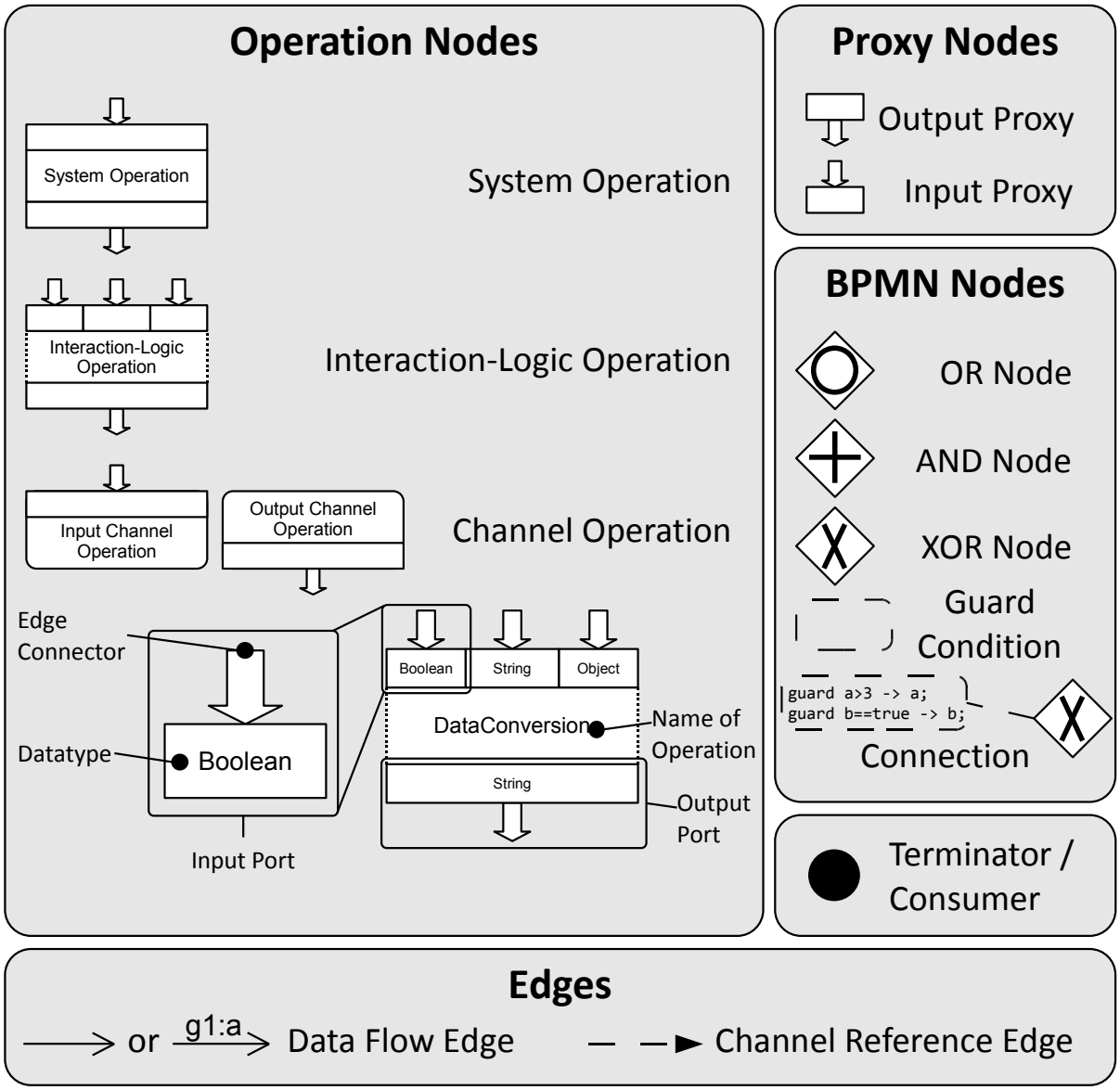


Figure 3.3.: VFILL: Visual specification and explanation

Figure 3.3 shows all VFILL items (nodes and edges) in five blocks, grouping the different items in the categories indicated by the headings. In the biggest block (upper left), all operation nodes can be seen as boxes with the associated ports indicated as smaller compartments in the operation node boxes. Thus, each input and output port is visualized as a box connected by an arrow to the port box (input port) or away from the port box (output port). A more detailed view of an operation is given in the lower part of the *Operation Nodes* block. The data type associated with each port is indicated by the label, as is the name of each operation.

For obvious reasons, types of operation nodes have to be differentiated from one another. The border of the operation box indicates whether it is a system operation (a solid border without rounded corners), an interaction-logic operation (a dashed border without rounded corners) or a channel operation (a solid border with rounded corners). In addition to labels, input-channel operations are differentiated from output-channel operations by the position of the rounded

corners. In the case of an input-channel operation, the rounded corners are at the bottom of the operation box, while in the case of an output-channel operation, the rounded corners are at the top of the operation box.

Proxy nodes are visualized in VFILL very similarly to port nodes because of their similar semantics, 'connecting something'. In contrast to ports, proxies are not connected or associated with an operation box. As indicated in their definition in FILL (cf. Definition 3.16), input proxies send data objects to a modeled interaction process as a result of an event occurring in physical representation of the user interface, for instance, after the user presses a button. Output proxies, on the other hand, consume data objects from the interaction process and send them to the interaction element associated with a certain proxy node. In general, one interaction process is associated with one interaction element. That is not explicitly specified by (V)FILL but should be understood as a statement for modeling VFILL. The implemented modeling tool, therefore, explicitly offers one modeling area or canvas for one interaction process for each interaction element.

The three *BPMN nodes* borrowed from BPMN modeling language are identically visualized as described in the BPMN specification [24, 197]. They are subsumed in the box under the one showing the proxy nodes. A BPMN node's outer shape indicates that this node is a BPMN node, and the sign inside indicates which kind of BPMN node it is. A circle indicates the OR node, a plus the AND node, and an X the XOR node. Guard conditions are connected with BPMN nodes by a dashed edge to differentiate them from data flow edges shown in the box at the bottom of Figure 3.3. Thus, guard conditions are shown in a dashed box with rounded corners to clearly differentiate them from visualizations of operations.

Furthermore, VFILL defines two types of *edges* that were defined above as elements of set E and relation c in Definitions 3.16 and 3.17. The first type is the standard edge, representing data flow that connects ports, proxies and BPMN nodes with one another. This kind of edge defines from which source to which destination a data object has to be sent. These edges can be decorated with labels that indicate the edge, the sent data, or both. When these labels are used as variables in guard conditions, they reference the sent data object. This is not the case, for instance, for a label at an outgoing edge of a branching BPMN node. Here, the label will reference the edge. The second edge type visualizes channels between input- and output-channel operations. This sort of edge is indicated by a dashed line instead of the solid one used for standard data-flow edges and a black arrow in contrast to the white arrow of a data-flow edge. Both types of edges are directed and 1-to-1 connections (no multi-edges), and arrows indicate their direction.

The last block under the BPMN Nodes block shows the *terminator node*, visualized as a big black dot. This visualization is identical or similar to the visualization used in UML activation diagrams [195] and BPMN.

Figure 3.4 provides an example of two main interaction processes, IP_1 and IP_2, modeled in VFILL. IP_1, which is surrounded by a dashed border, models the data processing of a String object that might, for example, result from an enter event of a text field on the physical representation. The entered String is sent to IP_1 as a data object in the moment in which the user presses the enter key. This piece of data is then sent to an AND node, which splits IP_1 into three sub-processes sending the same String to all of them. That is the default behavior of a branching AND node without a guard condition attached, which is optional in this case. The left sub-process generates an Integer value, using an interaction-logic operation to transform Strings to Integer values. The concrete transformation algorithm that is implemented in a

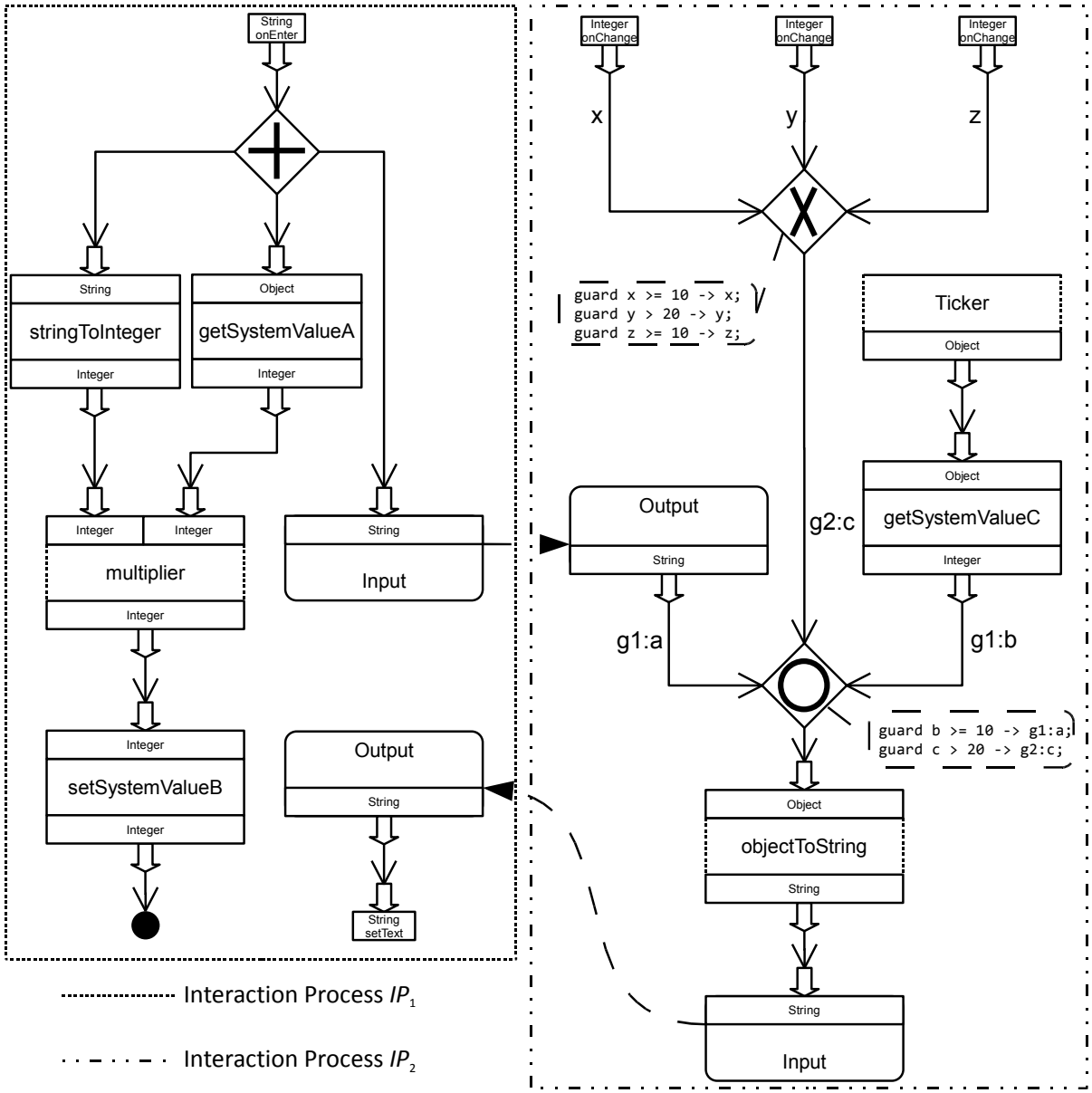


Figure 3.4.: VFILL: Example of a possible VFILL graph modeling several interaction processes

higher programming language is not directly viewable on this level of modeling VFILL. Still, adding additional information to this operation in a more complex visual editor or by adding the code in text form would help in this situation. The calculated result of this interaction-logic operation is passed to another interaction-logic operation that multiplies two incoming Integer objects. The other input value for this interaction-logic operation is passed from a system operation triggered by the String object in the mean sub-process. In this context, the String object's exclusive function is to trigger a system operation in the sense of node activation (cf. the above discussion concerning edge and node activation). The result of the multiplication operation is then passed to a system operation, setting this value to the system value B. The returned value from this operation is consumed by a terminator node.

The third sub-process (the right one in IP_1) activates an input-channel operation that sends the String object to IP_2, as shown on the right side of Figure 3.4 and indicated by a dashed-

dotted frame. The String is sent to an OR node via the edge labeled $g1 : a$, which is activated if the edge with inscription $g1 : b$ has also been activated by an data object. This happens if the system operation `getSystemValueC` is activated by an incoming data object. Here, this incoming object is generated and sent to the system operation by an interaction-logic operation *Ticker*. This operation sends a simple data object into the interaction process in clearly defined time steps, for example, every second. If a data object is sent via edge $g1 : a$ and $g1 : b$ the edge group $g1$ will be activated and will activate the OR node and its guard conditions will be evaluated according to the above-defined evaluation routine. The activation of edge groups is generally identical to the activation of nodes. If all edges of a group that is identified by the label in front of : are activated, the group is activated and acts similarly to one activated edge on a connected BPMN node.

Coming back to IP_2, when the OR node is activated, its associated guard condition will be evaluated. Then, the data object sent via edge $g1 : a$ will be sent to the non-labeled outgoing edge only if the data object b sent via edge $g1 : b$ is equal to or greater than 10. This is defined by the first guard condition. If group $g2$ is activated by sending a data object through edge $g2 : c$, the OR node will also be activated. In this case, the second guard condition will be evaluated. If the sent data object c via edge $g2 : c$ is bigger than 20, it will be transferred to the outgoing edge of the OR node. This data object sent by edge $g2 : c$ results from an XOR node in the upper part of IP_2. This XOR node is activated, if exactly one of the incoming edges x, y, or z is activated by an event occurring on the physical representation of the user interface. Depending on the guard conditions, the values are either transferred to the outgoing edge or not. If an XOR node fuses interaction processes, the guard condition has to be interpreted as follows. First, the target reference indicated by the arrow -> has to be interpreted. In IP_2, the guard shows three conditions. Here, the target reference defines to which incoming edge a guard condition is associated and which data object will be sent to the outgoing edge, depending on which condition can be evaluated to true. If one incoming edge is activated, the fitting condition will be derived from the target reference and evaluated, and if applicable, the corresponding data object that is also identified by the target reference will be sent to the outgoing edge. For instance, if edge x is to be activated, the first guard condition will be evaluated such that, if x is greater than 10, x will be sent to the outgoing edge. Thus, any single guard condition of an XOR node is associated by the target reference with one incoming edge, thereby defining the condition's association and which data object will be sent to the outgoing edge. Nevertheless, it would be possible to define more than one condition per incoming edge by using its reference in several guard conditions resulting in a non-deterministic selection behavior. The first condition evaluated to true will activate the outgoing edge, either there is another 'true' condition or not.

If the OR node in IP_2 is activated after evaluation of its guard condition, a data object is sent to its outgoing edge. This data object is sent to an interaction-logic operation, where it is transferred to a String object. The String object is then sent to an input-channel operation that is associated with one output-channel operation that is part of IP_1. This output-channel operation in turn sends the String object to an input proxy associated with the interaction element associated with P_1, the text field from the beginning.

Figure 3.4 is only a simple example that could easily be extended. Still, one point is missing: formal semantics for FILL. The above explanations give an idea of how the dynamic of a FILL model looks during runtime. The next step is the formal, algorithmic transformation of FILL graphs to a representation as reference net, offering formal semantics. Based on a simulation engine like Renew [150, 152], a FILL graph also becomes executable.

3.4. Transformation to Reference Nets

In addition to other reasons, the transformation of FILL to another formal language is motivated by its lack of formal semantics. Defining this kind of transformation to a formal language has other important advantages. First, the new formalism will be based on a formalism that is grounded in a scientific community and used in various application areas and thus, has been thoroughly investigated. Second, the use of Petri nets offers a huge set of verification and validation tools and a good theoretical background without the need to reinvent to wheel. Last, there are various tools for modeling Petri nets as well as many existing algorithms for simulating Petri net-based models to make them executable in real application scenarios.

There are several reasons for choosing Petri nets. A first look at FILL shows that this language is a graph-based language modeling data-processing processes. Therefore, any language FILL is transformed into should also be graph-based in order to prevent errors in transformation and make the creation of transformation algorithms easier to implement. Thus, the choice of non-graph-based formalisms, such as grammar-based approaches (for instance graph grammars [72], Lindenmayer grammars [222], etc.), is not appropriate. Unlike these formalisms, graph transformation systems or transition systems are a better choice because both are based on graph structures. Graph transformation systems are often powerful formalisms but also present problems in their theoretical basis, such as decidability, as described by Betrand et al. [26]. Thus, it is better to use a less powerful formalism that is still powerful enough to model the processes described by FILL. Therefore, classic transition systems seem to be the right choice for a successful transformation of FILL to a formalism providing formally specified semantics and a well-defined syntactic structure.

The next question is which concrete formalism to choose from the set of transition system formalisms. One well-known group of formalisms is probabilistic transition systems, such as Bayesian networks [124] or neuronal networks [234]. It is obvious that such formalisms are not suitable to use for a transformation of FILL, which is not a probabilistic. Still, in the context of building hybrid formalisms, these probabilistic approaches are of interest for adding further knowledge to modeled interaction logic. Another group of formalisms is the automatons used for defining languages in theoretical computer science [115]. Automatons were developed mainly for defining languages by accepting (or not) a sequence of terminal symbols. The terminal symbols were read in a sequence contrary to the parallel and non-deterministic data processing defined by FILL. This does not mean that it would be impossible to model interaction logic with automatons; however, doing so would make it more difficult and less accurate to transform parallel FILL processes into sequential automatons. Furthermore, FILL handles more complex (data) objects then terminal symbols. In the end, automatons are not suitable for the approach being used in this work because they provide less tool support for use in real applications. Especially in the context of tool support, only one type of formalism stood out as highly compatible: Petri nets. Petri nets allow the combination of non-deterministic and parallel processes in one model; moreover, there is an active research community associated with them that supports a broad variety of tools for modeling, simulation, and analysis in various use cases and scenarios. Many tools are available in the form of libraries that can be easily used in third-party implementation. Therefore, Petri nets are the right choice because they are able to reproduce the semantics of FILL and offer broad tool support paired with a solid theoretical background embedded in a lively research community.

Against this background, this section will introduce reference nets. Reference nets are a

special type of Petri nets that offer a suitable formalism for representing interaction logic on an appropriate level for formal simulation, verification, and validation. Reference nets are higher-level Petri nets that are able to handle complex data objects. Furthermore, they are supported by a solid implementation for modeling and simulation called *Renew*.

Reference Nets

Reference nets are a special type of Petri nets introduced in the PhD thesis by Olaf Kummer in 2002 [149]. In the following section, every formalization and definition are taking from this thesis to stay close to the original work by Kummer. As first step, some reasons for selecting this kind of Petri net will be discussed by describing alternatives and their pros and cons. In a next step, reference nets will be introduced formally to provide a basis for the formal transformation of FILL graphs to reference nets.

Motivation for Reference Nets

Before starting to argue for one or another Petri net-based formalism, some requirements arising from FILL and its transformation should be discussed in closer detail.

1. FILL describes data flow and data conversion processes; interaction logic is a collection of interaction processes that are data flow-based processes. Therefore, a formalism should offer a way of describing data or, more precisely, typed data objects and their processing. For data processing, it should be possible to connect processing formalisms, like a programming language, to the formalism, enriching the data processing by defining and adding *elementary* operations.

2. For implementation reasons, a simulator should be implemented for the formalism to run in an overall simulation of physical representation, interaction logic, and the system to be controlled.

3. Interaction logic is connected to a system on one side and to a physical representation of a user interface on the other. Thus, a formalism should offer a way of referencing physical elements and the system interface, for instance, through a programming language. This also addresses the need to introduce some kind of language to the basic formalism, a further requirement in this list.

4. To represent communication channels defined in FILL, the formalism should offer a way to formalize such constructions.

5. Nevertheless, the formalism should offer formal semantics for FILL. Therefore, the formalism should offer available and well defined formal semantics.

These requirements even more argues for the use of a Petri net-based formalism as it has been former discussed above. Still, a Petri net is a modeling language for use with concurrent processes in distributed systems [21, 216, 220]. Therefore, concurrent processes for data processing can easily be modeled using Petri nets. Furthermore, Petri nets have been investigated for over 45 years beginning with Petri's dissertation in 1962 [216]. During this time, many variations of the basic Petri net formalism have been developed and investigated on a formal basis in theoretical computer science research. Similarly, various modeling methodologies, validations,

and verifications have also been developed and investigated over time, for instance, in system engineering [98]. Moreover, many tools and simulators have been implemented to execute Petri nets in various ways in different contexts, as a list of tools shows[1].

Next, an applicable Petri net formalism has to be identified for the transformation of a FILL graph, especially in order to meet the need of a connection to other, programmatic formalisms and the formalization of channel operations. A further requirement from the above list is the need for typed data in Petri nets to influence the selection of the 'correct' type of Petri net. The following classification written by Monika Trompedeller in 1995 [268] offers a handy overview of Petri nets. It is based on a survey by Bernardinello and Cindio from 1992 [25]. It does not try to give an overview of up-to-date research results or to be complete, but it briefly summarizes the whole area of Petri net-based formalisms in the form of a brief taxonomy. The classification defines three different levels of Petri net-based formalisms, which are separated mainly by their expressiveness.

Level 1: Petri nets on Level 1 are characterized by places that represent Boolean values; thus, a place is marked by at most one unstructured token. Typical examples are:

- Condition/event systems
- Elementary net systems
- 1-safe systems
  - State machines

Level 2: Petri nets on Level 2 are characterized by places that can represent integer values; thus, a place is marked by a number of unstructured tokens. Typical examples are:

- Place/Transition nets
  - Ordinary Petri nets

Level 3: Petri nets on Level 3 are characterized by places that can represent high-level values; thus, a place is marked by a multi-set of structured tokens. Typical examples are:

- High-level Petri nets with abstract data types
- Environment relationship nets
- Well-formed (colored) nets
- Traditional high-level Petri nets
  - Colored Petri nets
  - Reference nets

This overview shows that FILL can most easily be transferred to a Petri net of level 3. This is due to the first requirement, that a suitable formalism provide a form of expression for complex data (types). The idea is to represent the data object sent from the physical representation to the system through the interaction logic, or vice versa, by a token in the net. A classical

[1] http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html

approach for such higher Petri nets has been proposed by Jensen [126]. He extends classical Petri nets by an inscription language that offers typed places and complex tokens representing data of a specific type. Often, Petri net-based formalisms are accompanied with a certain tool support for simulation and verification. In the case of colored Petri nets, Jensen's group implemented the tool CPNTools[2] [128] to connect visual modeling and verification tools.

Still, neither the transformation and representation of connections between the net, the physical representation of a given user interface, and the system nor the formalization of channel operations is easily supported by Jensen's CPNs. Therefore, it would be helpful to introduce reference mechanisms to the Petri net formalism. This leads to the choice of *reference nets*. Reference nets are level 3 nets offering complex, typed tokens [155] and a mechanism for referencing the (Java) code of a higher programming language and synchronous channels [43], which offer a well-defined way of sending tokens between net instances. There is also tool support for reference nets for the implementation of this work. Renew[3] is a simulator for reference nets implemented in Java; it enables calls between the net and the Java source code [150, 152]. Thus, complex functionality can be introduced to interaction logic modeled as a reference net, a system interface implemented in Java can be called, and a physical representation, such as Java Swing [164] implementation, can be referenced. These implementation issues are discussed further in Chapter 5. There are probably other options for implementing a simulator for reference nets or Petri nets in general, such as logical or functional programming languages [7, 139]. Still, it is not the aim of this work to implement a new simulator especially because Renew is a solid implementation offering easy access to and from Java code to integrate a simulated reference net into a whole program structure. This fact in particular is one of the most important arguments for the decision to use reference nets as the formalism used to transform FILL, as argued above.

In the following section, a short and informal introduction to reference nets will be given based on the work of Kummer [149], to show features of reference nets, and thereby their connection to FILL.

Introduction to Reference Nets

This section will give a brief and informal introduction to reference nets. The reader is assumed to have the basic knowledge of Petri nets required for understanding this section. Books by Reisig [232], Baumgarten [21], and Priese [220] can be recommended as introductory reading about Petri nets, although they do not constitute a complete list. In his books [125, 126, 127], Jensen describes colored Petri nets as extensions of basic S/T nets by using complex structured tokens. An overview of various types of Petri net formalisms, especially time-dependent and stochastic approaches to Petri nets, can be found in [22]. Best and Fernandez give an overview of various Petri net terminologies in [27]. Petri Net World is an online platform giving an overview of the Petri net community and of various Petri net types and tools [301].

Reference nets were introduced by Olaf Kummer in his PhD thesis [149] in 2002. He delineates three major concepts which extend basic colored Petri net definitions: (a) synchronization of different transitions in a net that are basically concurrent and independent of each other, (b) creation of more than one instance of a net, and (c) extension of possible token values by references to other net instances. He introduces these three major concepts in three steps, using an example that has been slightly adapted to fit the context of modeling interaction logic.

[2] http://wiki.daimi.au.dk/cpntools/cpntools.wiki
[3] http://www.renew.de/

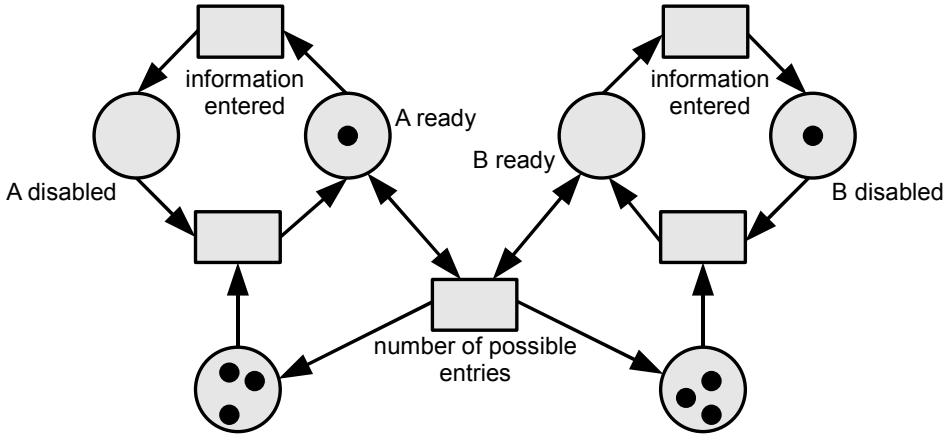


Figure 3.5.: Simple example for a S/T net describing the behavior of two interdependent text fields in a simple user interface

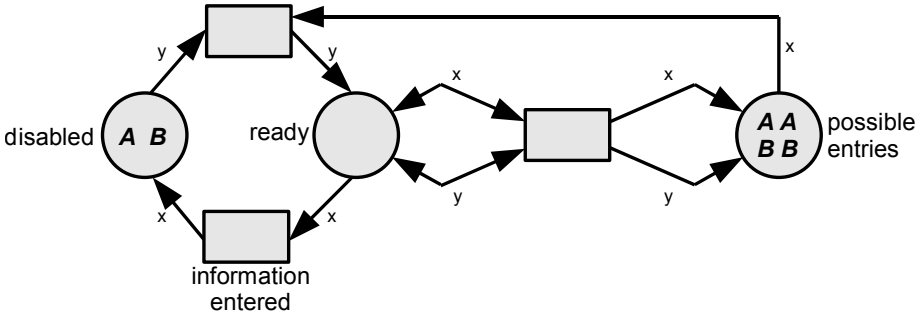


Figure 3.6.: Extension of the example of simple interaction logic using colored Petri nets

From S/T Nets to Colored Nets The example the various extensions will be introduced by is that of a simple interaction logic describing how the status of two text fields A and B of the physical representation of a given user interface could change between *ready*, where the user can enter information, and *disabled*, where the user cannot enter anything in the text fields. The disabled status of a text field, whereby an interaction element is grayed out, is well known from standard user interfaces. A further restriction on entering information is a value that defines how many times information can be entered. Every time information has been entered, the text field should be disabled. It should only be possible to set a new number of entries if both text fields are in ready mode. A possible S/T net (level 2 net) modeling this interaction logic can be seen in Figure 3.5.

The structure of interaction logic for both text fields is very similar. Jensen [126] solves this problem of multiple similar structures in a net by adding structured tokens to the basic S/T net definition. This makes it possible to represent the state of both subnets with only one net and two tokens representing the text fields. The resulting net is shown in Figure 3.6. Figure 3.7 shows a part of this net to give an idea of how the rule for firing transitions in these nets is defined. If a transition fires, any variable at an incoming edge has to be bound to a token from the incoming place that is itself connected to that edge. The binding of values to variables is decided by unification algorithms known from mathematical logic studies [12, 145, 244]. This will also be the case for reference nets and their simulation tools. If it is not possible to bind variables via unification, the transition cannot fire and is therefore inactive. In the example,

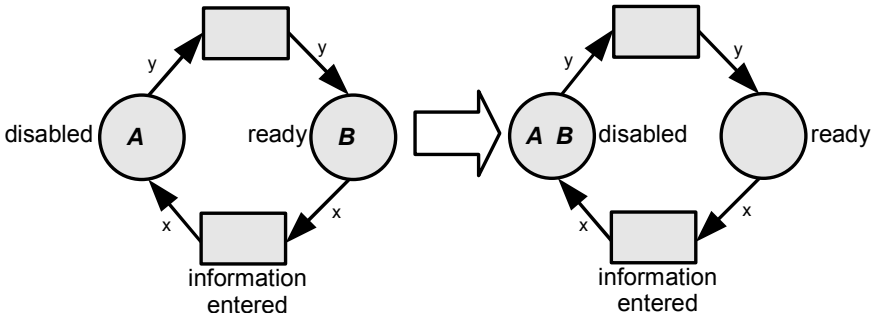


Figure 3.7.: Switching of a transition in a colored Petri net through binding complex values to edge labeling variables

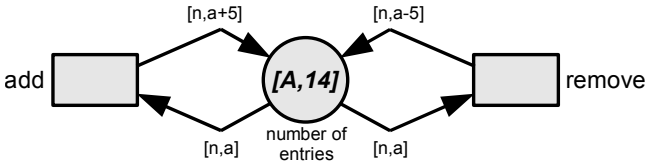


Figure 3.8.: Representation of a possible implementation of interaction logic for a slider defining the number of entries

the lower `information entered` transition can fire through a binding of token B to x. Because the variable at the outgoing edge is also x, a new B token is first bound to x at the outgoing edge and then added to the outgoing place. The result is shown on the right in Figure 3.7.

In Jensen's understanding of colored Petri nets, places are multi-sets of structured tokens as shown in Figure 3.6. In this example, the number of possible entries in a specific text field is represented by a token indicating its owner. Text field A has two further entries free. The same is true for text field B. Another possible way of representing the number of entries is excluding a part of the net to form a separate one. The resulting net could be connected to the interaction element that defines the possible number of entries while both text fields are in ready state, for instance, a slider defining the new number of entries allowed. This would be congruent with the convention of creating one interaction process for every interaction element. A model of this separation is shown in Figure 3.8 as a net offering two operations represented as transitions: (a) `add` and (b) `remove`. Here, the structured token is a tuple of two items. The first item represents the owner of the number of entries; the second item shows the number of possible entries, here 14. The `add` transition increases the number of possible entries by 5; the `remove` transition decreases the number of possible entries by 5. For instance, if a slider is used to input the number of entries, every tick to the right would fire the `add` transition (in Figure 3.8) and similarly, every tick to the left would fire the `remove` transition, decreasing the possible number of entries.

From Colored Nets to Nets with Synchronous Channels One major problem of the example shown in Figure 3.8 is the mixture of the explicit dependency of text fields A and B and their implicit dependency on another interaction element defining the number of entries. It seems to be more natural to differentiate the model of explicit dependencies from that of implicit dependencies as can be seen in Figure 3.9. In this extension, some transitions have to be associated with each other. This association is indicated by dashed lines shown in Figure 3.9.

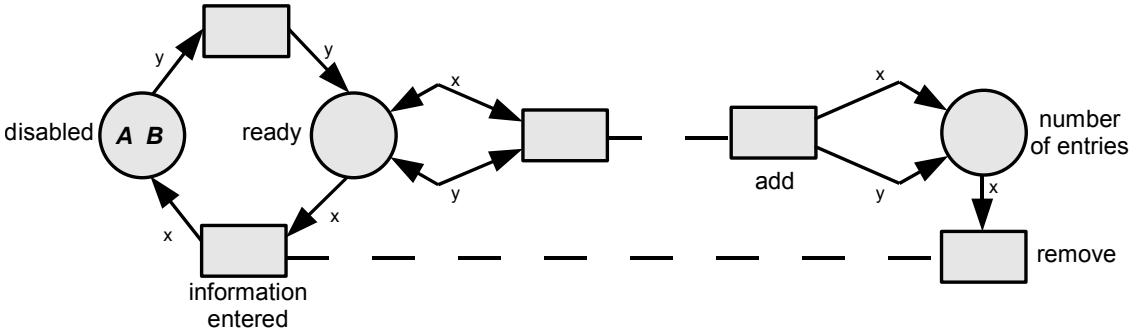


Figure 3.9.: Separation of text filed's behavior and their restrictions through the number of possible entries

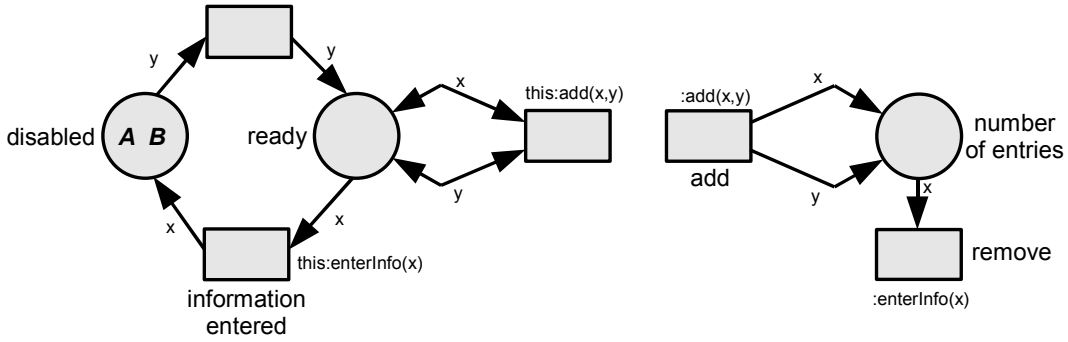


Figure 3.10.: Textual addresses mark synchronous channels

To avoid a change in the fire semantics in the net, these transitions have to be associated in a very strict way. If one transition fires, the associated transition has to be fired instantaneously, that is, in a synchronous way. This semantic behavior can be compared with function or method calls in higher programming languages. As can be seen in Figure 3.10, this kind of synchronous calling mechanism can be added to a Petri net formalism using a specific kind of inscription. A synchronization is called by, for instance, the inscription `this:enterInfo(x)`. This inscription associates its transition with the transition inscribed `:enterInfo(x)`. The variable `x` is likewise bound to the value of the incoming token as is the case for variable labeling edges. The bound value is then transferred to the called transition as a parameter and then brought to the partial net representing a part of the implicit dependency. This concept of synchronous channels was first introduced by Christensen and Hansen [43] and was transferred to and used in Kummer's reference net formalism [149].

Synchronization can also be used to describe the parallel calls of transitions. This means that more than one synchronization can be modeled through one channel. This can be seen in Figure 3.11. The `add` channel is called twice. This kind of model results in a double firing of the `add` transition with two different values. Using synchronous channels, it is now possible to introduce the simple model of the slider's interaction process from Figure 3.8 to the example shown in Figure 3.11. The resulting net is shown in Figure 3.12.

In contrast to basic colored nets, nets with synchronous channels have the ability to talk about simultaneous actions and not only about concurrent or serial actions [149].

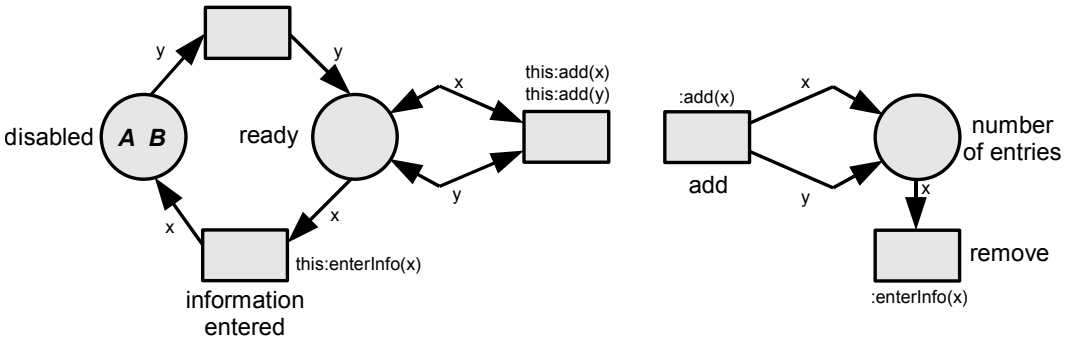


Figure 3.11.: Re-use of synchronous channels

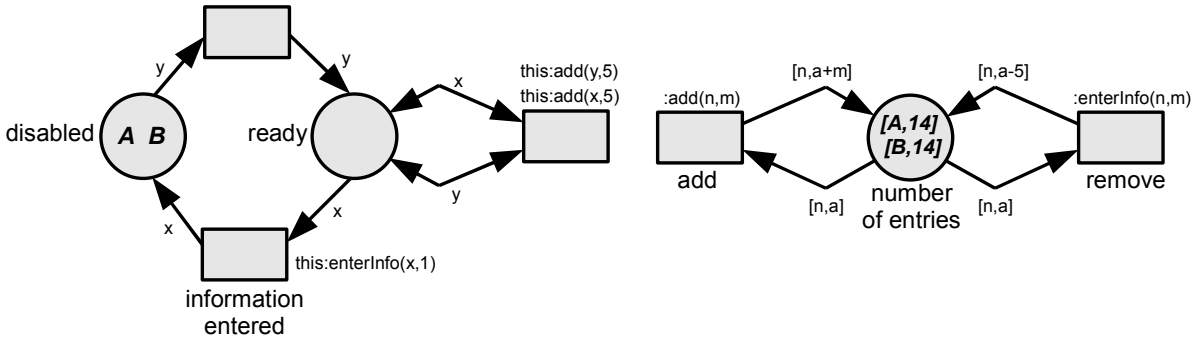


Figure 3.12.: Complete separation of text field's behavior and interaction process of slider defining the number of entries

Reference Nets Classic Petri nets and higher Petri net formalisms normally limit the existence of a token to a singleton. During runtime there is a singleton token in exactly one net. Kummer lifts this restriction by allowing multiple instances of a net. Furthermore, he allows references on net instances or exemplars to be represented as tokens in a net instance. This ability distinguishes reference nets from colored nets with synchronous channels.

In Figure 3.13 there are two net patterns on the left: the `creator` net and the `refnet` pattern. In the `creator` net, a transition creates two net instances of the `refnet` pattern, and binds them to the variables `x` and `y`. On the right in Figure 3.13, the `creator` net is shown during runtime. Here, the two instances of `refnet` were created by the firing of the upper transition labeled with expression using the `new` operator for instantiation. After the lower transition has fired, the references will be consumed or deleted, but the net instances of the `refnet` will still exist. All net instances thereby are independent (as is true for higher, object-oriented programming languages), with the exception of synchronous channels.

This referencing mechanism allows modeling of system reorganization instead of only description of the changes in system states as in standard colored nets through the token game. Using net instances in modeling reduces the complexity of net patterns, offering better understanding of complex models.

In conclusion, reference nets offer the following abilities, which distinguish them from other, more basic Petri net-based formalisms:

1. complex, colored tokens based on object-oriented systems like that of Java programming language,

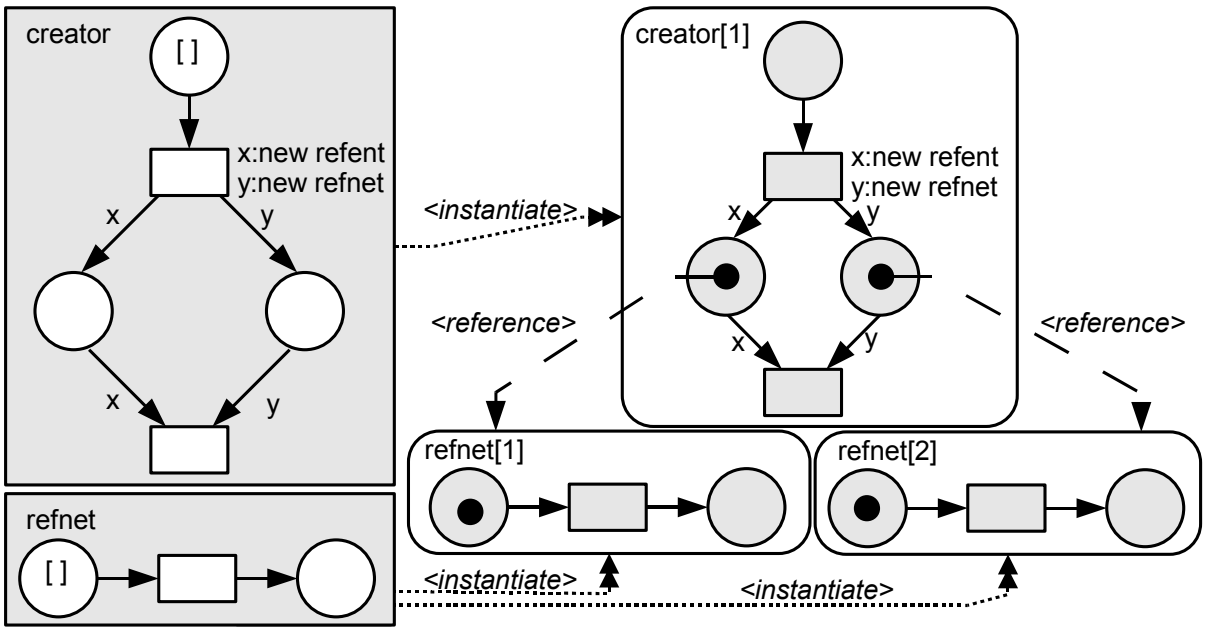


Figure 3.13.: Example of two net patterns (shown on the left with a gray background color) that are instantiated as two net instances (shown on the right)

2. synchronous channels that describe synchronous firing of distributed transitions in a net, and

3. the use of referencing net instances based on net patterns.

The following section will introduce the formal definition of reference nets as a basis for the transformation of FILL graphs to reference nets.

Formal Definition of Reference Nets

One motivation for the transformation of a FILL graph to a reference net is the lack of a formal semantics for FILL. According to the definition of the formal transformation to reference nets, the semantics of reference nets is also suitable for FILL. In addition, the resulting transformed reference net can be simulated in Renew, as well as validated and verified using various tools for the validation and verification of Petri net-based models.

A thorough definition of the formal semantics of reference nets is complex and will not be recapitulated here in its entirety. For a closer look at the complete definition, see Kummer's work [149, p. 107–119], which outlines the definition of formal semantics for reference nets in detail.

In brief, the formal semantics of reference nets is based on a conversion of a reference net to a graph transformation system. The token game of the reference net is transferred to a graph, in which a node represents a token and its inscription indicates the place in which the token is positioned. Net instances are represented as special type of node that are inscribed with the net instance's name. Edges that refer a token (node) to a net instance (node) represent a token (node) referencing a certain net instance (node). Edges that refer a net instance (node) to token (node) indicate the belonging of this node to this specific net instance. The fire event of a

transition is modeled based on the single-pushout approach known from the theory of categories described, for instance, by Ehrig et al. ([73, 77]) as transformation rule.

The definition of semantics for FILL and the formal transformation of FILL to reference nets require a formal definition of reference net formalism. Before the structure of reference nets can be defined, a formal definition of the labels and inscriptions of the reference net's elements, like places, transitions, and edges, has to be described. That is of primary interest, because a lot of semantic information results from a complex inscription language associated with a basic Petri net definition like Jensen's [125]. Kummer [149] defines such a language on the basis of labeled algebra as discussed by Ehrig et al. [76]. Labeled algebra associates any element of the algebra with a word from a given alphabet. It also guaranties that the result of a calculation with an operator is the concatenation of the words of the arguments. The definition of labeled algebra is based on signatures that are necessary for the definition of algebra for multiple sorts that can be compared with types in object-oriented programming languages.

For the following definitions, detailed discussions and descriptions, as well as a fundamental derivation of the formal reference definition can be found in Kummer's dissertation [149], as well as in his publications, like [146, 148, 150]. Especially, the mechanism of synchronous channels has been described in detail in these resources, based on works of Christensen et al. [43]. The work at hand will give only a short overview of the most important definitions and formalization necessary for formal transformation of FILL to a representation as a reference net.

Definition 3.19 (Signature) *A pair $Sig = (S, Op))$ is a* signature *if S is a set of sorts and Op is an indicated set of sets of operators that are pairwise disjoint. $Op_{w,s} \subseteq Op$ with $w \in S^*$ and $s \in S$ denoting the operators with arguments of sort w and a result of sort s.*

$$op : s_1 \times \ldots \times s_n \rightarrow s$$

can be written for $op \in Op_{s_1 \ldots s_n, s}$. ◇

Based on Definition 3.19, which defines the term signature, a special type of algebra called *Sig-Algebra* can be defined as follows:

Definition 3.20 (Sig-Algebra) *If a signature $Sig = (S, Op)$ is given, a pair $A = (S_A, Op_A)$ is called* Sig-Algebra *or simply* Algebra*, if $S_A = \{A_s\}_{s \in S}$ is an indexed set of pairwise disjoint sets and $Op_A = \{A_{op}\}_{op \in Op}$ is an indexed set of functions, where for $op \in Op_{s_1 \ldots s_n, s}$ and $s_1, \ldots, s_n, s \in S$, $A_{op} : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ is true.*

Furthermore, it is defined that $Set(A) = \bigcup_{s \in S} A_s$. If there is no confusion, $x \in A$ will be written for $x \in Set(A)$. ◇

By introducing an extending labeling function l, a basic Sig-Algebra (cf. Definition 3.20) can be extended to a labeled algebra, as follows:

Definition 3.21 (Labeled Algebra) *A signature $Sig = (S, Op)$ and an alphabet X are given. A* labeled Algebra *is a triple $A = (S_a, Op_A, l)$, where (S_A, Op_A) is a signature and l is a total function with $l : S_A \rightarrow X^*$, such that for $s_1 \ldots s_n \in S$, $s \in S$, $op \in Op_{w,s}$, and $x_1 \in A_{s_1}$ to $x_n \in A_{s_n}$,*

$$l(A_{op}(x_1, \ldots, x_n)) = l(x_1) \ldots l(x_n)$$

is true. ◇

A labeled algebra associates any element of an algebra with a word from a given alphabet guaranteeing that the result of an operator is the concatenation of its words given as parameter to the operator. This type of definition for an inscription language extended by the definition of super-signatures (cf. Definition 3.22) makes it possible to define a highly flexible declaration language for reference nets without concreting the language too early and thereby reducing portability between algebraic systems.

Definition 3.22 (Sub- / Super-signature) *Two signatures* $Sig = (S, Op)$ *and* $Sig' = (S', Op')$ *are given.* Sig *is a* Sub-signature *of* Sig'*, if* $S \subseteq S'$ *and for* $w \in S^*$ *and* $s \in S$, $Op_{w,s} \subseteq Op'_{w,s}$ *is true.*

Sig' *is also called a Super-signature of* Sig. ◇

Based on these definitions of *signature* and *labeled algebra*, the reference declaration as basis for talking about variables and types can now be defined as follows:

Definition 3.23 (Reference Declaration) *A* reference declaration *is a 9 tuple*

$$D = (Sig, A, Sig', s_R, Q, V, v_0, \tau, C),$$

where

- $Sig = (S, OP)$ *is a signature,*
- $A = (S_A, Op_A)$ *is a Sig-Algebra or Data-Algebra,*
- $Sig' = (S', Op')$ *is a super signature of* Sig*,*
- $s_R \in S - S'$ *is a distinguished sort of reference,*
- Q *is a set of net identifier with* $Q \cap Set(A) = \emptyset$*,*
- V *is a set of variables,*
- $v_0 \in V$ *is a distinguished variable,*
- $\tau : V \to S'$ *is a typing function, and*
- C *is a set of names for channels.* ◇

Definition 3.23 does not specify any limitations on the algebra A. Up to this point, there is no necessity to reduce the possible range of inscription languages resulting from reference declarations. Later on, it will be shown that this algebra can be matched to the Java type system which will also be used for FILL and all implementation aspects in the context of this work.

Q is a set of net identifier where any element $q \in Q$ will be associated with a concrete net pattern. Based on this identifier and the associated net pattern, a new net instance of this net pattern can be created. v_0 is a special identifier that always points to a net instance in which a transition fires. This kind of identifier can also be seen as a `this` or `self` pointer. In general, only net instances are differentiated, and there is only one net sort.

Based on this formal definition of a reference declaration, reference nets can be formally defined as follows [149, p. 218]:

Definition 3.24 (Reference Net) *A reference net declaration*

$$D = (Sig, A, Sig', s_R, Q, V, \tau, C)$$

is given. A reference net is a 7 tuple

$$N = (S, T, K, k, G, U, c)$$

with

- *sets S of places, T of transitions and C of global defined channels, where S, T and C are pairwise disjoint,*
- *K a set of edges and channels,*
- *the function*

  $$k : K \to (S \times T \times V \times \{in, test, out\} \cup C \times T \times V \cup C \times T \times V \times V),$$

  which assigns edge and channel objects to an applicable tuple concerning their type,
- *$y \in K$, with $k(y) = (c, t, v, v')$, where $c \in C$ and $\tau(v') = s_R$,*
- *$G : T \to (Op' \times V * \times V)$ a set of equality specifications, where*

  $$(op, v_1 \dots v_n, v) \in G(t) \text{ and } op \in Op'_{\tau(v_1)\dots\tau(v_n),\tau(v)},$$

- *$U : T \to \mathcal{P}(\mathcal{V} \times \mathcal{V})$ a set of inequality specifications, such that for $(v, v) \in U(t)$, $\tau(v) = t(v)$ is true, and*
- *$c : T \to (V \to Q)$ associating a partial function to every $t \in T$ with*

  $$\forall v \in dom(c(t)) : \tau v = s_R. \qquad \diamond$$

The definition of reference nets differs in some aspects from other higher Petri net formalisms. The first difference is that edges are inscribed only with variables and not with complex terms. Based on the equality specifications, these variables can be easily extended to complex terms if necessary. In the context of this dissertation, this will not be necessary.

A second difference is that places are not typed. This aspect is not a problem for the transformation of FILL to reference nets. The type-safeness of FILL will be integrated by edges inscribed by typed variables based on Java's type system.

Complex guard conditions like those defined in other Petri net formalisms are replaced here by equality and inequality specifications. If complex guards are necessary as in the transformation of BPMN nodes, the data algebra used has to be extended in such a way that complex (Boolean, assertion logical) conditions can be described. By using the Java type system as the data algebra, the resulting reference net offers complex guard conditions compatible with the simulator that is implemented in Java, which can evaluate these expressions.

K mainly describes edges in the net that can have different types. Therefore, an element $x \in K$ is associated with a quadruple $k(x) \in S \times T \times V \times \{in, test, out\}$ defining the neighboring place $s \in S$, the neighboring transition $t \in T$, the edge's inscription $v \in V$, and its type as elements of the set $\{in, test, out\}$, whereby *in* means that the edge is pointing to the transition and *out* means the edge is pointing to the place. In addition to *in* and *out* edges, *test* edges can also be defined. Test edges assign firing conditions to the connected transitions with the difference that if the transition fires, the tokens of the connected place are not removed (cf. [126]). Thereby, an edge of type *test* influences the firing of a transition, but does not change the connected place's marking. Still, this type of edge is not necessary for the transformation

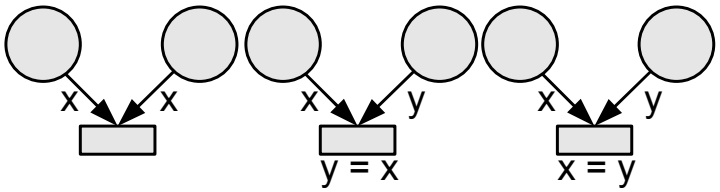


Figure 3.14.: An example of using '=' in a unification context in contrast to the use as assign operator in Java

of FILL to reference nets in this version. Therefore, this type of edge is not further described or investigated.

$x \in K$ can also be a synchronous channel. If $k(x) \in C \times T \times V$, the channel is called an *uplink*. An uplink indicates that the associated transition can receive "invocations". If $k(x) \in C \times T \times V \times V$, the channel is called a *downlink* and represents an invocation of another transition that can be placed in another net instance.

Java as Inscription Language

For implementation and modeling reasons, one concrete language should be used as an inscription language and data algebra. For reference nets, Kummer used Java for implementation and in this context, Java is part of the reference net's inscription language, which is introduced informally in [149], based on [147] and [151]. The following paragraphs draw from explanations in [149] to offer a close and informal description of the inscription language used as a Java derivate to avoid loss of information.

Equality and Assigning The '=' operator is in Java declared as an assign operator with various limitations that are in conflict with the general unification approach of variables in Petri nets. In Petri nets, variables are unified with values during simulation or runtime. This difference between the two approaches will be explained using the example in Figure 3.14 (cf. [149, p. 247]). All three nets have the same semantics. In the net on the left, both incoming edges of the transition are inscribed with the variable x, which means that the tokens or values on the incoming places must have the same value and type or that their type is a super- or subtype of the other. Only if this requirement is fulfilled, can both variables be bound to these identical values and the transition fires.

The net in the middle of Figure 3.14 shows the use of the '=' operator as a condition for unification with the same semantics as described for the example on the left. The same is true for the example on the right, which only shows that the '=' operator in the reference net's unification algorithm (for binding variables to values) is commutative. This contrasts with the assigning operator in Java, where on the left side a variable and on the right side a value has to be assigned or defined.

This kind of *re-definition* of the '=' operator in reference nets leads to following conventions for the '=' operator concerning equality:

1. Values are bound to variables through unification algorithms and not by the '=' operator.
2. On both sides, complex expressions concerning the equality condition for unification can be written, like $x + 1 = y + 1$, which is not allowed in Java.

3. Concerning types in Java, only the left variable can be a super type or the same type as the right variable to prevent testing for types during runtime. For reference nets, it is only important that one variable is a super type of the other; there is no position restriction concerning the '=' operator.

In summary, the assign operator in Java is used in reference nets as an equality operator in the mathematical sense for controlling the unification process.

Guards In addition to the unification that controls the enabling and firing conditions of transitions, additional inscriptions for transitions are introduced that *guard* the transitions. These guard conditions are closely related in their semantics to *guard conditions* in FILL. A guard condition is an inscription of a transition that is identified by the keyword `guard` followed by a proposition-logical expression. The transition can fire only if this expression is evaluated to `true`. Variables in these conditions are also bound by unification algorithms as is the case for general variable binding in reference nets. Guard conditions are restricted to Boolean expressions based on propositional logic as will be defined for FILL below. In general, guards are evaluated after a matching binding has been found for edges' inscribed variables and the given equality conditions.

Actions As described above, assigning values to an object or variable in the sense of Java's semantics is not possible using the '=' operator. '=' is reserved for proving equality during unification. That is why a new keyword is helpful for marking sections of a transition inscription that is independent from the unification process. To be independent from unification means that such expressions can be interpreted as normal Java expressions. Doing so makes it possible to use the '=' operator in the Java sense and thus for assigning values to variables independently of the unification process.

The keyword marking these sections to transitions' inscriptions is `action`. Expressions following `action` are interpreted according to Java type rules and expressions. In addition to the use of '=' in the Java sense, as is now possible, `action` expressions are only executed during the firing of transitions. To this end, various requirements have to be met. For example, the `action` inscription has to be used if Java methods are called from reference nets. Further examples are the transformed subnets resulting from various FILL nodes and the applied transformation algorithms that will be presented in the next section. More about `actions` can be found in [149, p. 249].

Synchronous Channels Channels are indicated by a name with a leading colon. The name is followed by a parameter list in round brackets. Variables on this list are separated by commas. Uplinks and downlinks are separated by leading identifiers of the net instance in which a *call* of a transition via a synchronous channel is executed. Here,

$$: channel(x, x, 20)$$

an example of an uplink is given, identified by the absence of a net identifier in front of the colon. The example shows only the channel name and the parameter list. The variable x has to be bound to a value before the transition associated with the inscription can fire. One possible matching downlink is

$$n : channel(a, b, c),$$

where n indicates the net instance, and the a, b and c variables have to be bound to values before *calling* the uplink in net n. A further requirement on the values bound to a, b and c is that they have to match the associated uplink. If the previously given uplink is called the downlink, the values of a and b have to be bound with identical values (through unification), and c has to be bound to the value 20. If this is not possible, neither the transition associated with the up- nor downlink can fire. The name of the channel *channel* indicates which transition in net n should be fired in the case of a valid binding. If there are more than one matching uplink in net n (that also fulfill the further restriction on the bindings), one of them is chosen non-deterministically.

If a downlink references an uplink in the same net instance, the keyword `this` can be used, such as

$$this : channel(a, b, c).$$

This type of downlink will mainly be used in transformed FILL graphs because up to now there is no use to the referencing mechanism. Still, there are future work issues that can benefit from this specific characteristic of reference nets. One aspect would be the modeling and implementation of multi-user systems based on formal interaction logic, where the dependencies between individual interaction logic models can be identified for every user. Also, a further net can model the interrelation between the user's interaction logics regarding aspects like task and responsibility dependencies. Furthermore, it would be possible to create interaction logic models describing various responsibilities of users which can then be replaced, depending on which user is controlling the system via the user interface. In this context, more than one user can also be logged in to the system with the same responsibility resulting in multiple instances of a net pattern describing this responsibility. A more specific use of the reference mechanism is for modeling and integrating interaction-logic operations not only as abstract concepts but as concrete net. Also, a system modeled as a reference net can be easily integrated into an existing interaction logic without changing the formalism. Although this is planned for future work, it offers further evidence that reference nets are the right choice for modeling formal interaction logic.

Tuples and Lists In some cases, single-valued variables (which can also be of a container type, like Vector or Hashmap) are not enough or make the structure of a reference net too complex. Therefore, reference nets introduce tuples and lists, whereby tuples are more important to the transformation of FILL than lists are. It should be assumed that the calculation of $(x-1)*(x+1)$ has to be modeled in a reference net. `N1` in Figure 3.15 is an incorrect example, while `N2` is a correct model, which uses a tuple. In net `N1`, the sub-results from tokens `1` and `2` can be mixed up. Using tuples, these sub-results are connected by the tuple that makes mixing up the sub-results impossible.

Lists instead represent organized selections of values without an upper boundary of the number of elements. Lists are syntactically defined using braces because in the inscription language of reference nets, blocks like those used in higher programming languages are not supported. This means that there is no confusion concerning using braces to represent lists instead of code blocks. More information about lists can be found in [149, pp. 260,261].

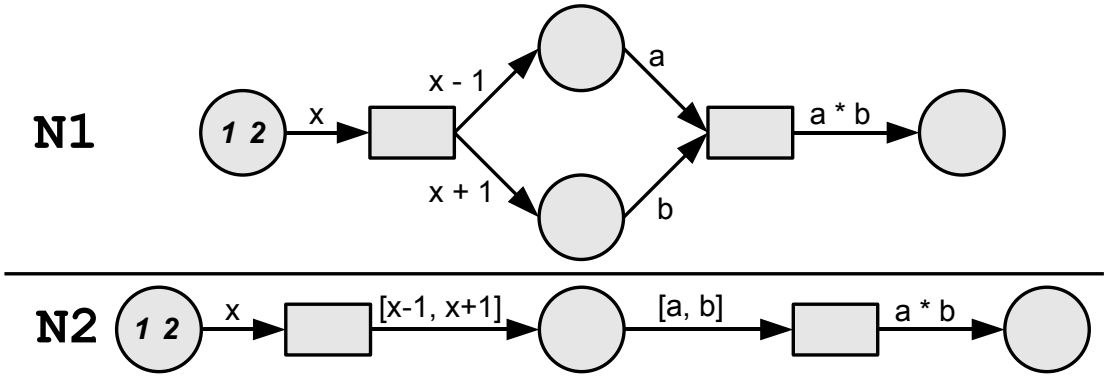


Figure 3.15.: Example of using tuples to avoid modeling errors like that shown in net `N1` by associating both sub-results in a tuple, as in net `N2` (cf. [149, p. 259])

Method Calls in Reference Nets to Java Code Communication between transitions is based on synchronous channels. This mechanism can be used for synchronous communication between transitions in a single net instance or in multiple instances as described above. The next step would be to extend the synchronous communication mechanism by a calling mechanisms to call methods in a programming language from the net and vice versa. If this extension is based on synchronous channels, the Java-based inscription language does not have to be further extended. Thus, the syntax for communication with Java methods is exactly the same as for synchronous channels. That makes sense because the semantics is more or less the same. The difference is that the uplink now calls a method in a Java class and does not match it with another transition's inscription in a given net instance. Downlinks are identically defined and inscribed to a transition as are synchronous channels and are now being called by Java code. This functionality is based on a stub architecture, which handles the net's communication with the loaded Java class during the runtime of the simulation. This stub-based architecture, paired with the simulation engine, was implemented in Renew, connecting Java classes with a simulated reference net.

Based on this (informal) declaration of reference nets' Java-like inscription language, which was introduced for Renew as a modeling and simulation engine for reference nets, the next section will introduce a specific inscription language for FILL graphs. Edge labels and guard conditions in particular have to be more clearly specified in order to make an algorithmic transformation to reference nets possible.

Edge Labels and Guard Conditions in FILL

Before starting to define formal transformation of FILL to reference nets, label and guard inscriptions for edges and BPMN nodes in FILL have to be formally defined. Guard conditions, in particular, are defined to conform with a reference net's declaration of guards based on propositional logic. This simplifies transformation and prevents different levels of expressiveness of both languages. The first step is to define edge labels in the form of an EBNF grammar [120], as in the following definition.

Definition 3.25 (Edge labels in FILL) *For all FILL graphs*

$$F = (S, I, C_I, C_O, P_I, P_O, X_I, X_O, B, T, P', E', l, g, c, t, \omega, \mathcal{L}, \mathcal{B}),$$

all words (labels) $l_i \in \mathcal{L}$ are words of language $\mathcal{L}$, which is defined by the following EBNF:

$$
\begin{aligned}
\text{LABEL} &= [\text{GROUP':'}]\text{VARIABLE};\\
\text{VARIABLE} &= \{\text{CHARACTER}\};\\
\text{GROUP} &= \{\text{CHARACTER}\};\\
\text{CHARACTER} &= \text{NUMBER} \mid \text{'a'} \mid \text{'b'} \mid \ldots \mid \text{'z'};\\
\text{NUMBER} &= \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'};
\end{aligned}
$$

◇

In Definition 3.25, edge labels are defined as words being composed of characters and numbers, where the colon ':' is a reserved character indicating edge groups. Groups are especially important for OR nodes as described in Section 3.3 (see Figure 3.4). Labels identify the interaction process and the data object that is passed 'through' the edge it is associated with. This semantics will be manifested in the transformation to reference nets later on. In Definition 3.26, a EBNF grammar will be defined for the guard conditions that are associated with BPMN nodes.

Definition 3.26 (Guard Conditions in FILL) *For all FILL graphs*

$$F = (S, I, C_I, C_O, P_I, P_O, X_I, X_O, B, T, P', E', l, g, c, t, \omega, \mathcal{L}, \mathcal{B}),$$

all words (guard conditions) $b_i \in \mathcal{B}$ are words of language $\mathcal{B}$, which is defined by the following EBNF:

$$
\begin{aligned}
\text{GUARD\_EXP} &= \{\text{GUARD';'}\};\\
\text{GUARD} &= \text{'guard' CONDITION ['->' VARIABLE]};\\
\text{CONDITION} &= (\text{CONDITION BOOL\_OP CONDITION}) \mid [\text{'!'}]\text{'(' CONDITION ')'} \mid\\
&\quad [\text{'!'}]\ \text{BOOL\_VAR} \mid \text{NUM\_COMP\_EXP} \mid \text{BOOL\_VAL};\\
\text{NUM\_COMP\_EXP} &= (\text{NUM\_EXP COMP\_OP NUM\_EXP}) \mid \text{'(' NUM\_COMP\_EXP ')'};\\
\text{NUM\_EXP} &= (\text{NUM\_EXP ARITH\_OP NUM\_EXP}) \mid\\
&\quad \text{NUM\_VAR} \mid \{\text{NUMBER}\};\\
\text{NUM\_VAR} &= \text{VARIABLE};\\
\text{BOOL\_VAR} &= \text{VARIABLE};\\
\text{VARIABLE} &= [\text{GROUP':'}]\{\text{CHARACTER}\};\\
\text{GROUP} &= \{\text{CHARACTER}\};\\
\text{ARITH\_OP} &= \text{'-'} \mid \text{'+'} \mid \text{'*'} \mid \text{'/'};\\
\text{BOOL\_OP} &= \text{' |'} \mid \text{'\&'};\\
\text{COMP\_OP} &= \text{'>'} \mid \text{'<'} \mid \text{'>='} \mid \text{'<='} \mid \text{'=='};\\
\text{CHARACTER} &= \text{NUMBER} \mid \text{'a'} \mid \text{'b'} \mid \ldots \mid \text{'z'};\\
\text{NUMBER} &= \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'};\\
\text{BOOL\_VAL} &= \text{'true'} \mid \text{'false'};
\end{aligned}
$$

◇

The semantics of a guard is also defined by its conversion to reference nets. Basically, the semantics of a guard is based on propositional logic extended by an arithmetic expression using numerical values connected by arithmetic operators. Through operators for comparing numerical expressions, they are transformed to Boolean values, which can be evaluated in a logical expression. As with reference nets, guard conditions in FILL are syntactically closely related to logical expressions in JAVA [244].

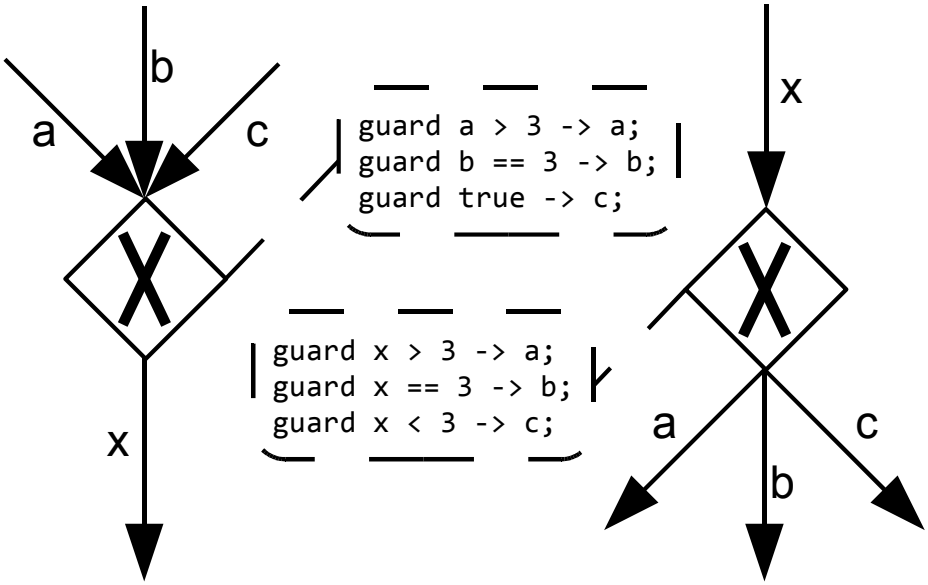


Figure 3.16.: Example of guard conditions in an XOR node in a FILL graph for both fusion and branching

The semantics of the stylized arrow -> is different from guards in reference nets. The arrow differentiates the (Boolean) condition from a reference. This *target reference* has different semantics depending on whether it is used in a branching or a fusion case of a BPMN node. Before specifying this in more detail, it should be remembered that an edge label can reference the edge or the data object that activates the edge. For instance, by using the edge label in guard conditions, the label is used to reference the data object activating the edge and not the edge itself.

Branching In branching, the target reference defines which outgoing edge will be activated with the incoming data object if the the Boolean expression it precedes can be evaluated to true. The target reference references an edge by using its label as a reference. Here, the target reference references an edge and not a data object.

Fusion The target reference defines which data object from which incoming edge will be sent to the outgoing edge if the preceding Boolean expression can be evaluated to true. Here, the target reference references a data object, in contrast to a target reference in branching.

For a better understanding, the following examples of different guard conditions and their evaluations will be described in detail. The first guard condition is an example of a branching XOR node as can be seen in Figure 3.16 on the right. Here, the XOR node's three outgoing edges are labeled a, b, and c, and its incoming edge is labeled x, as well as it guard condition is given by

$$\begin{aligned} &guard \quad x > 3 \quad -> a; \\ &guard \quad x == 3 \quad -> b; \\ &guard \quad x < 3 \quad -> c; \end{aligned}$$

This guard condition and its evaluation can be described as follows. If the incoming value referenced by x is greater than 3, the edge labeled a will be activated by sending value x to

edge a. Where x is equal to 3, edge b will be activated by sending value x to it. If x is less than 3, edge c will be activated by the value x. Here, the target references define which outgoing edge is to be activated by the incoming value to the BPMN node after a positive evaluation of the associated condition. It should be mentioned at this point that nondeterministic conditions can also be created. For instance, if the third condition in the above example is changed to

$$guard \quad x <= 3 \quad -> c;$$

it will be evaluated differently. If x is exactly 3, the value would be sent to edge b or edge c, because both expressions would be evaluated as `true`. For the transformation to reference nets, this is unproblematic. In this aspect, reference nets are nondeterministic, as is the case for all Petri net-based formalism.

In cases of fusion, as can be seen on the left in Figure 3.16, the evaluation is a bit different from the above example. This is because the target references are now associating a condition with a certain incoming edge. This means that, if an edge labeled a is activated by an incoming value, the first condition will be evaluated. In this case, if the incoming value of edge a is greater than 3, it will be sent to the outgoing edge without further evaluation of other conditions in the guard expression. Here, label a has two different meanings: (1) It references the edge as a target reference, and (2) it functions as variable that is bound to the data object that is sent 'through' the edge. The guard condition

$$\begin{aligned} guard \quad a > 3 \quad -> a; \\ guard \quad b > 3 \quad -> b; \\ guard \quad true \quad -> c; \end{aligned}$$

can be described as follows. If the incoming edge labeled a is activated, the first condition referencing a, and no other, will be evaluated. Thus, if the data object sent via edge a is greater than 3, the value of variable a is sent to the outgoing edge. If edge b is activated, the second condition will be evaluated. In this case, the outgoing edge will also be activated if the data sent via edge b is greater than 3. Data sent through edge c is always sent to the outgoing edge; the condition is always `true`.

A special type of edge labels is grouped labels. Such labels are only important for edges associated with OR nodes. This node type can fuse or branch groups of edges and not only either single (XOR node) edges, or all incoming or outgoing edges (AND node). Groups are defined using the reserved colon string in an edge label, as in Figure 3.17. This example shows, fusion and branching in an interaction process using an OR node. In general, the OR node is activated only if all the edges of a group are activated. In fusion, the OR node is activated if edges $g1 : a$ and $g1 : b$ of group $g1$ are activated or if edge $g2 : c$ of group $g2$ is activated. The individual labels a, b and c are still important to identify the individual edge of a group or its associated data objects. In fusion, a possible guard expression

$$\begin{aligned} guard \quad (a == 3)\&(b > 5) \quad -> g1 : a; \\ guard \quad true \quad -> g2 : c; \end{aligned}$$

can be defined. If group $g1$ is activated through the activation of both edges $g1 : a$ and $g1 : b$, the first guard condition will be evaluated. The outgoing edge will be activated if a is equal to 3 and b is greater than 5. The value of edge $g1 : a$ will be sent to the outgoing edge following

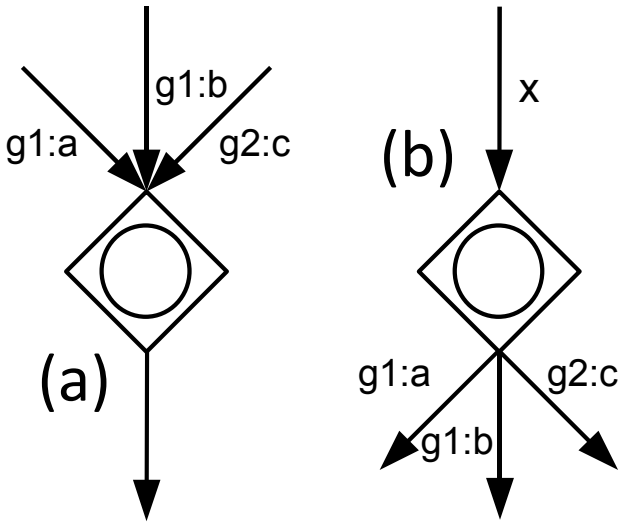


Figure 3.17.: Example of group labels for branching and fusion of an interaction process using an OR gateway: (a) shows a fissioning OR gateway; (b) shows a branching OR gateway using groups

a positive evaluation of this condition. Here, it can be seen that in fusion the target reference always references the data object by using the individual label of edge $g1 : a$. The second guard condition states only that if edge $g2 : c$ is activated, it will directly activate the outgoing edge and by sending its data object, referenced by c.

In branching, outgoing edges are referenced as groups because a whole group of edges will always be activated if the preceding condition is evaluated to true. A possible guard expression for the given OR node

$$\begin{aligned} &guard \quad x > 3 \quad -> g1; \\ &guard \quad x <= 3 \quad -> g2; \end{aligned}$$

could be defined. If the incoming data object x is greater than 3, group $g1$ will be activated. Activation of a group of edges means that edges $g1 : a$ and $g1 : b$ will be activated with the same data object referenced by x. If x is less than or equal to 3, group $g2$ will be activated by sending the data object referenced by x to edge $g2 : c$. Nondeterministic guard expressions are also possible in this situation using group edges. This is due to the fact that the conditional part is defined similarly to the former cases; only the target references change to accommodate grouping.

Restrictions on the use of guard conditions for BPMN nodes in FILL have to be made for semantic and transformation reasons. Table 3.2 shows these restrictions for all BPMN nodes in FILL. AND nodes only accept exactly one guard condition in fusion excluding a target reference. That results from the semantics of an AND node. Thus, all outgoing edges will be activated by the incoming data object. In this case, the guard condition only controls whether the incoming data object will activate the outgoing edges or not. In OR and XOR nodes, for any outgoing group or edge, a guard has to be specified to indicate which incoming data object will activate which outgoing group or edge. In fusion, all guards only control which data object is sent to the outgoing edge. If guard conditions are optional, it is possible not to specify a guard condition. In this case, a random-selection heuristic decides which data object activates

Table 3.2.: Conditions to adding guard conditions to BPMN nodes and further restriction for the use of references in guard conditions

Node Type	Branching	Fusion
AND	(optional) ONE guard WITHOUT target reference	(optional) ONE guard WITH target reference
OR	(obligatory) ONE guard WITH target reference per OUTGOING GROUP	(optional) ONE guard WITH target reference per INCOMING GROUP
XOR	(obligatory) ONE guard WITH target reference per OUTGOING EDGE	(optional) ONE guard WITH target reference per INCOMING EDGE

the outgoing group or edge.

On the basis of the reference nets introduced, their formally defined syntax, and the specification of label and guard condition syntax for FILL graphs, the next section introduces the transformation of FILL graphs to reference nets by specifying transformation algorithms.

Transformation of FILL Models into Reference Nets

Transformation of FILL graphs to reference nets will be discussed from two different perspectives: (a) a visual representation of the various conversion rules will give an overview of how FILL graphs are converted to reference nets accompanied by (b) a formal description of the conversion of a FILL graph-based interaction logic to a reference net given as an algorithm in pseudocode.

Conventions

For a formal transformation, the following conventions and extensions of the given definitions have to be introduced based on a given FILL graph:

- If $op \in S \cup I \cup C_I \cup C_O$ is a given operation, than $P_I^{(op)} \subseteq P_I$ is an indexed set of n input ports with
$$P_I^{(op)} = \{p_i | p_i \in P_I, \exists (p_I, op) \in P'\},$$
where index i identifies the i-th input port of operation op.
$p_O^{(op)} \in P_O$ is the output port of operation op, such that
$$\exists (p_O^{(op)}, op) \in P' : p_O^{(op)} \in P_O.$$

- f is a function, with
$$f : S \cup I \cup C_I \cup C_O \to \mathcal{F},$$
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
$$\kappa : X_I \cup X_O \rightarrow \mathcal{I},$$
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
$$id : S \cup I \cup C_I \cup C_O \cup P_I \cup P_O \cup X_I \cup X_O \cup B \rightarrow \mathcal{ID},$$
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 (discussed below), ids play an important role.

- := represents an assigning operator that makes it possible to re-bind a set to a 'new' set in an iterative process or algorithm. This allows mathematical sets to be extended as in a programming language, as the example
$$E := E \cup \{a, b, c\}$$
shows. This expression has the same semantics as would various programming languages. The set E is extended by the elements a, b and c. Here, E is treated more like a variable that is rebound to a union of two sets.

- To simplify the association of inscriptions to transitions based on G and U as equality and inequality specifications, which are themselves based on Java's type and logic system, the function $inc : T \rightarrow \mathcal{IN}$ associates a transition with an inscription based on the inscription language informally introduced in the previous section. $\mathcal{IN}$ represents a set of all possible inscriptions based on the language and formal specification of G, U and c in the reference net definition. Thus, $i \in \mathcal{IN}$ is a String representing a binding condition using equality, a channel identified by a colon (up- or downlink), a guard condition, or an action.

- $m_s : P_O \rightarrow S$ and $m_t : P_I \rightarrow S$ are functions, where P_O is the set of output ports and P_I is the set of input ports of a FILL graph and S is a set of places in a reference net and NOT a set of system operations. m_s matches an input port to its corresponding place in the reference net; m_t matches an output port to its corresponding place in the reference net. This convention simplifies the definition of transforming a data edge as described below.

- Any string in typewriter font type, such as `a`, indicates the data 'a' of type String. The combination of Strings with mathematical symbols implies the use of a adequate String representations of this mathematical notation as String in the transformed reference net. For example, in case of a variable v_0 the resulting String would be `v0`.

Still, a basic convention for inscriptions in reference nets is necessary. This convention will specify the type of data objects that are passed as tokens through the net. Based on the tuple data structure in reference nets, a convention for transformation is defined as follows. In the definition, any edge inscription in the transformed reference net must take the form of a tuple, such as

```
[procID, ieID, DataObject].
```

The variables have the following meanings:

`procID` This variable defines an *id* for the interaction process, which is modeled using the net. The main purpose of this id is to enable some simplification of the resulting net. `procID` is of type String.

`ieID` This *id* identifies the original interaction element that triggers the interaction process. `ieID` is also of type String.

`DataObject` This is the data object that is originally generated by the input interaction element, or the system to be controlled and is modified by any operation that it passes.

The function of `procID` and `ieID` will be explained in greater detail in the discussion on transforming operations, proxies, and BPMN nodes below.

Preconditions

Beginning the transformation, $N = (S, T, K, k, G, U, c)$ is an empty reference net with a given reference declaration $D = (Sig, A, Sig', s_R, Q, V, \tau, C)$, which satisfies the convention introduced by Java as an algebraic data type system, as well as its formalisms for transition inscriptions described in Section 3.4. Additionally, F is a (not empty) FILL graph that has to be transformed to N using an iterative transformation algorithm. This algorithm will be described as elementary transformation steps for any FILL element. The use of conflicting symbols will be clear in the context of use transforming F to N. Otherwise, further information will be given.

Transformation of System and Interaction-logic Operations

Transformations of system or of interaction-logic operations are, for the most part, identical, differing only in the number of input ports. For system operations, exactly one input port is transformed (as shown in Figure 3.18), while for interaction-logic operations, 0 to n input ports are transformed. Still, each type of operation has exactly one output port that is transformed. Figure 3.19 shows the transformation of an operation node with three input ports. The transformation follows two main rules:

1. Every port is transformed into a place in the reference net. Depending on the type of port, this place is the source (input port) or the destination (output port) of an edge connected to the transitions that represent the operation.

2. Every operation is transformed into two transitions. One represents the operation call, shown in Figure 3.18 with four parameters, which is connected to edges the sources of which are places representing input ports. The other transition represents the return value of the operation and is therefore connected to the place representing the output port by an edge that points from the transition to the (output port) place.

The following pseudocode defines a part of the transformation algorithm extending the given (at the beginning empty) reference net N and transforming a given system operation s from F. The transformation of all system operations from F to N will be done by embedding the following algorithm into an iterative process over all elements in F, which will be described at the end of this section. This is true for each node in F, no matter what kind of FILL node it is. This is also true for data edges, where channel edges are transformed during the transformation

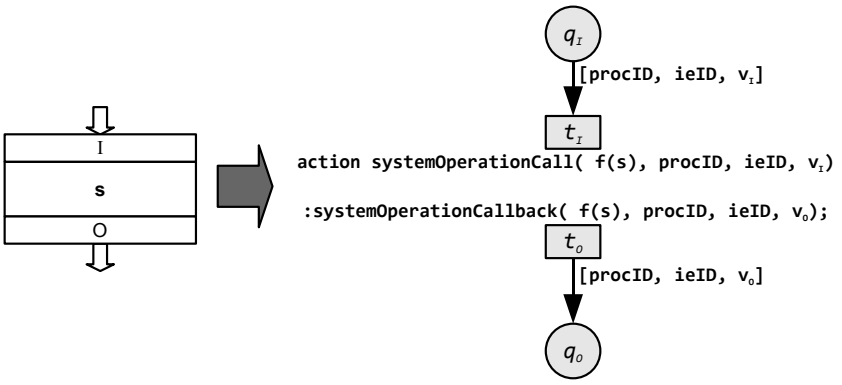


Figure 3.18.: Transformation of a VFILL system operation in a subnet of a reference net

of channel operations. The type of pseudocode used in Algorithm 3.27, describing the transformation of system operations, will be used identically in other transformation algorithms for other FILL elements.

Algorithm 3.27 (Transformation of System Operation) For a given system operation s from F, the transformation to a subnet of N will be generated.

(1) $v_I \in V$ is a variable, where $\tau(v_I) = t(p_I)$ with $p_I \in P_I^{(s)}$.

(2) $v_O \in V$ is a variable, where $\tau(v_O) = t(p_O^{(s)})$.

(3) t_I and t_O are transitions, where $t_I, t_O \notin T$,
$inc(t_I) =$ `action systemOperationCall(`$f(s)$`, procID, ieID,` v_I`)`, and
$inc(t_O) =$ `:systemOperationCallback(`$f(s)$`, procID, ieID,` v_O`)`.

(4) q_I and q_O are places, where $q_I, q_O \notin S$, $m_t(p_I) = q_I$, and $m_s(p_O) = q_O$.

(5) u_I and u_O are tuple variables of structure
$u_I =$ `[procID, ieID,` v_I`]`, and $u_O =$ `[procID, ieID,` v_O`]`.

(6) e_I and e_O are edges, where $e_I, e_O \notin K$,
$k(e_I) = (p_I,\ t_I,\ u_I,\ in)$, and $k(e_O) = (p_O,\ t_O,\ u_O,\ out)$.

(7) $T := T \cup \{t_I, t_O\}$; $S := S \cup \{q_I, q_O\}$; $K := K \cup \{e_I, e_O\}$.

In the first step of Algorithm 3.27, variables are defined with types conforming to the types of input and output ports in s. In the second step, two transitions are defined with inscriptions conforming to the up- and downlink declarations for synchronous channels in reference nets (introduced above) and their ability to connect to Java source code as a method call. Which system operation has to be called on the Java level is encoded by $f(s)$, which defines the concrete system operation or method. This will also be the case for interaction-logic operations, as described in Algorithm 3.28. For the edges defined in the last step of Algorithm 3.27, tuple variables are defined, which include the formerly defined variables. The algorithm finishes by adding all the generated elements to certain sets defining the reference net N. The given system operation s and the resulting subnet from Algorithm 3.27 can be seen in Figure 3.18.

The transformation of an interaction-logic operation s_{il} from F is slightly different from the transformation of a system operation. An interaction-logic operation transformation is described by Algorithm 3.28, which differs from Algorithm 3.27 in how it handles input ports. Interaction-logic operations have 0 to n input ports, which all have to be iteratively transformed.

Algorithm 3.28 (Transformation of Interaction-logic Operation) For a given interaction-logic operation s_{il} from F, the transformation to a subnet of N will be generated.

(1) $n = \#P_I^{(s_{il})}$.

(2) $v_i \in V$ are variables, where $\tau(v_i) = t(p_i)$ with $p_i \in P_I^{(s_{il})}$, $i = 0, \ldots, n-1$.

(3) $v \in V$ is a variable, where $\tau(v) = t(p_O^{(s_{il})})$.

(4) q_O is a place, where $q_O \notin S$, and $m_s(p_O^{s_{il}}) = q_O$.

(5) t_I and t_O are transitions, where $t_I, t_O \notin T$ with
$inc(t_I) =$ `action ilOperationCall(`$f(s_{il})$`, `$id(s_{il})$`, procID, ieID, `v_0`,..., `v_{n-1}`)`,
$inc(t_O) =$ `:ilOperationCallback(`$f(s_{il})$`, `$id(s_{il})$`, procID, ieID, `v`)`.

(6) u_O is a tuple variable of structure $u_O =$ `[`$id(p_O^{s_{il}})$`, ieID, `v`]`.

(7) e_O is an edge, where $e_O \notin K$, and $k(e_O) = (q_O,\ t_O,\ u_O,\ out)$.

(8) $T := T \cup \{t_I, t_O\}$; $S := S \cup \{q_O\}$; $K := K \cup \{e_O\}$.

(9) **FOR** $i = 0$ **TO** $n-1$ **DO**

- **(a)** q_i is a place, where $q_i \notin S$ and $m_t(p_i) = q_i$.
- **(b)** u_i is a tuple variable of structure $u_i =$ `[procID, ieID, `v_i`]`.
- **(c)** e_i is an edge, where $e_i \notin K$ and $k(e_i) = (p_i,\ t_I,\ u_i,\ in)$.
- **(d)** $S := S \cup \{q_i\}$; $K := K \cup \{e_i\}$.

In Algorithm 3.28, variables are defined for each of the input ports of s_{il} as is one variable for its output port. Similarly to Algorithm 3.27, two transitions are defined that represent calls to and from Java code implementing the functionality of the interaction-logic operation. Here, the method name used indicates that the corresponding operation is an interaction-logic operation, so that it is impossible to confuse this operation with a system operation. The concrete interaction-logic operation that has to be called on the Java level is encoded similarly to system operations, using the function $f(s_{il})$. The **FOR**-loop at the end of Algorithm 3.28 generates the necessary places and edges to represent the input ports of for s_{il} in N and adds them to the sets S and K of N. Figure 3.19 visualizes the transformation that results from Algorithm 3.28 using an example of an interaction-logic operation with three input ports.

Before examining the transformation of other FILL nodes, several issues concerning the transformation of interaction-logic operations should be mentioned. First of all, calling external methods in this way can generate problems when conducting a formal analysis of interaction logic that relies on approaches and techniques such as those used in process analysis based on Petri net formalism. If only the consistency of used data types is important for the analysis,

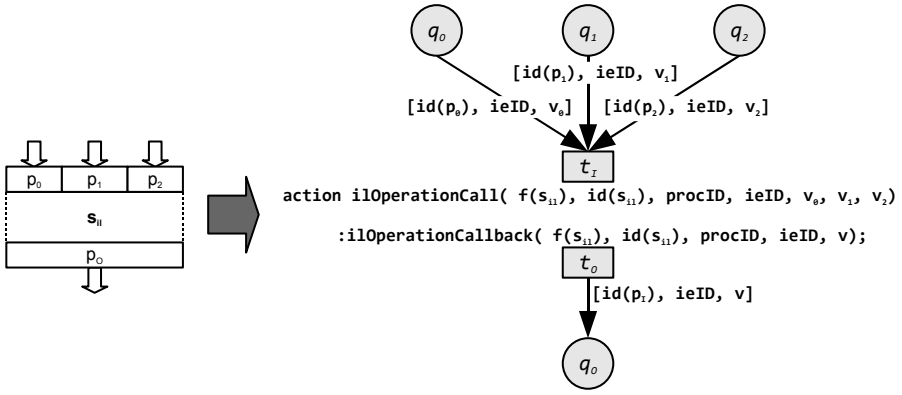


Figure 3.19.: Transformation of a VFILL interaction-logic operation in a subnet of a reference net

there will not be a problem. However, a problem in analysis arises if the type of functionality implemented by the interaction-logic operations has to be taken into account because the analysis method would have to take other formalisms into account (such as Java) beside the reference net formalism. Here, two solutions to this problem are mentioned:

Use of One Transition with Data Transformation given as an Inscription In this solution, the interaction-logic operation is not transformed into two transitions, but into one transition that is inscribed by a data transformation. How complex this transformation can be is limited by the inscription language. Simple conversions can be described by such means as arithmetic operations and type conversions. Still, the exclusive use of inscription is not as expressive as the Java-based approach.

Use of Two Transitions Referencing a Further Net Instance In this case, the two-transition concept is maintained but changed in that the downlink transition creates a net instance to represent the interaction-logic operation and sends the data to it. Therefore, the interaction-logic operation models complex behavior as would a reference net while doing data conversion or other operations. Figure 3.20 provides an example of this approach as it would be modeled using Renew. On the left side, there is a cutout of more complex interaction logic; this represents the transformed interaction-logic operation. The two transitions representing the interaction logic operation are not connected via a synchronous channel to Java code but to a net instance of net `ggtNet`. The net pattern of net `ggtNet` can be seen on the right side of Figure 3.20. The net instance of net `ggtNet` will be generated by the initialization part of the interaction logic net on the left (which will be also generated by an extended transformation algorithm). Any time the interaction logic operation is activated by two values being sent to the input places, the transition inscribed `n:ggt(this,v0,v1)` can fire, resulting in firing the `:ggt(n,y,z)` transition in the net instance `n` of type `ggtNet`. The parameters of the downlink transition refer to the interaction logic net (using `this`) and to the two parameters sent to the interaction-logic operation. The net reference is necessary for a callback with the resulting value, which is, in this case, the greatest common divisor (ggt) of the values `v0` and `v1`.

Based on the latter approach, various types of complex algorithms for data conversion can be included in the interaction logic without the necessity of introducing a higher-level language.

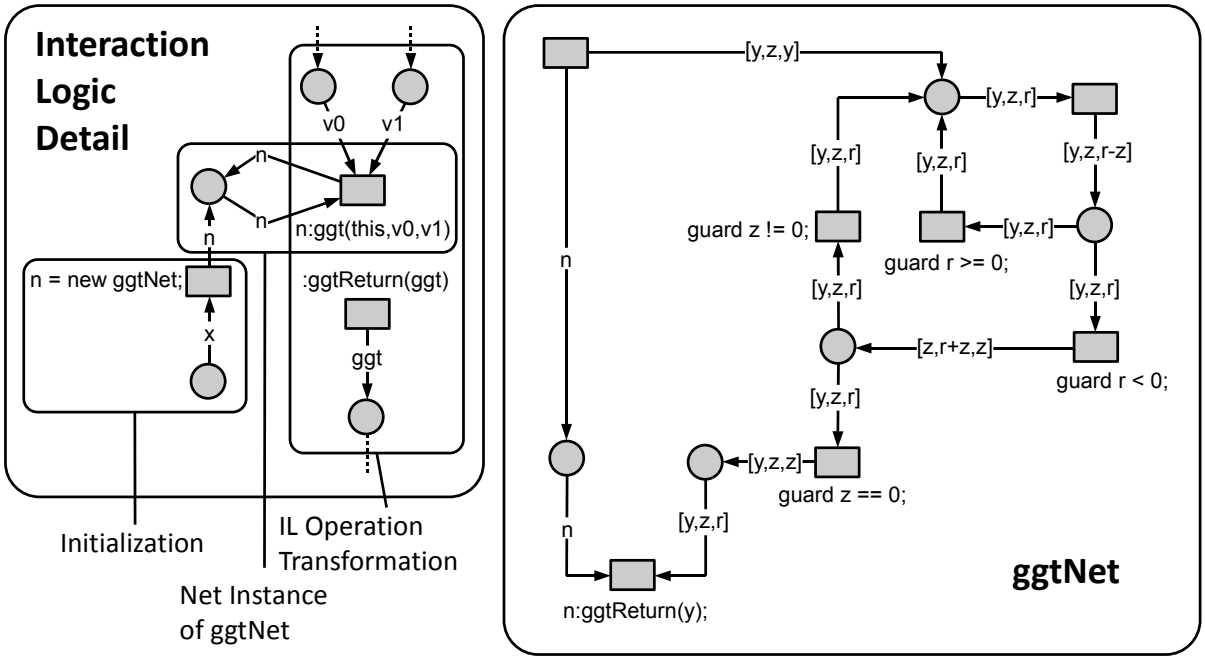


Figure 3.20.: Example of possible extension to the transformation algorithm for interaction-logic operations using the referencing mechanism formalized and implemented for reference nets; modeled in Renew

The same is true for system operations. Here, one net instance of the system implementation as a reference net is initialized and passed to all system operation representations in the interaction logic. A precondition for this approach is that the system be modeled as a reference net. If this is done and all interaction-logic operations are modeled as reference nets as well, the whole model can be analyzed using the analysis tools and algorithms for Petri nets.

A further matter that will be of even greater interest in the context of the next section should also be mentioned here. If the system and interaction-logic operations are modeled as reference nets, applied reconfigurations to interaction logic can also be directly integrated into the system's model if necessary. This is of interest in situations in which the user makes decisions that were not implemented or considered beforehand, for instance, when the user changes a critical system state back to a normal system state. By applying this approach, the system is able to learn from human behavior or, more specifically, from the reconfiguration of interaction logic conducted by a human user.

Transformation of Proxies

Transformation of proxies is slightly different than transformation of operations. A proxy represents a value sent from the physical representation of a user interface (output proxy) or a value sent to the physical representation in order to present a visualization (input proxy). Figure 3.21 shows the transformation of an output and an input proxy to a reference subnet. In such cases, the transition represents the import of a value into the net if it is the source of an edge pointing to a place or the export of a value out of the net if the transition is the destination of the edge pointing to the corresponding place. The place represents the point of connection for the rest of the net to the subnet representing the proxy. Algorithm 3.29 shows the transformation of an output proxy *op* from F.

Algorithm 3.29 (Transformation of Output Proxy) For an output proxy op from F, the transformation to a subnet of N will be generated.

(1) v is a variable, where $\tau(v) = t(op)$.

(2) t is a transition, where $t \notin T$, and $inc(t) =$ `:`$\kappa(op)$`(`v`)`.

(3) q is a place, where $q \notin S$, and $m_s(op) = q$.

(4) u is a tuple variable of structure $u =$ `[`$id(q)$`,` $\kappa(op)$`,` v`]`.

(5) e is an edge, where $e \notin K$, and $k(e) = (q,\ t,\ u,\ out)$.

(6) $T := T \cup \{t\}$; $S := S \cup \{q\}$; $K := K \cup \{e\}$.

The transformation Algorithm 3.30 of an input proxy ip is more or less identical to the transformation of an output proxy, as was shown in Algorithm 3.29.

Algorithm 3.30 (Transformation of Input Proxy) For an input proxy ip from F, the transformation to a subnet of N will be generated.

(1) v is a variable, where $\tau(v) = t(ip)$.

(2) t is a transition, where $t \notin T$, and $inc(t) =$ `action widgetCallback(`$\kappa(ip)$`,` v`)`.

(3) q is a place, where $q \notin S$, and $m_t(ip) = q$.

(4) u is a tuple variable of structure $u =$ `[`$id(q)$`,` $\kappa(ip)$`,` v`]`.

(5) e is an edge, where $e \notin K$, and $k(e) = (q,\ t,\ u,\ in)$.

(6) $T := T \cup \{t\}$; $S := S \cup \{q\}$; $K := K \cup \{e\}$.

The results of Algorithms 3.29 and 3.30 can be seen in Figure 3.21. Both algorithms transform a proxy to a subnet consisting of a place, a transition, and an edge, where the edge's direction indicates whether the subnet represents an input or an output proxy. Especially important for Algorithms 3.29 and 3.30 is the correct transformation of the connection to the physical element that is represented by these proxies. This problem is solved by ids referencing the interaction elements in the physical representation layer. These ids are associated with proxies using the function id. In the process of implementation, ids can be generated during runtime and thus do not have to be part of the formalism. For output proxies, the id of the associated interaction element is simultaneously the channel inscription of the corresponding transition in the transformed subnet. With output proxies that call Java methods, an interface method has to be declared called `widgetCallback`. Both decisions make sense from the perspective of an implementation for simulating formal interaction logic. Here, every interaction element knows its id and thus which transition has to be called. In callback, the necessity that any extension of the physical representation would result in a complex extension of the programming interface makes a common definition of the one-interface method reasonable. Which interaction element will be referenced is defined by the first parameter of the interface method. More information concerning implementation and simulation will be presented in Chapter 5, which introduces the framework implemented for this dissertation.

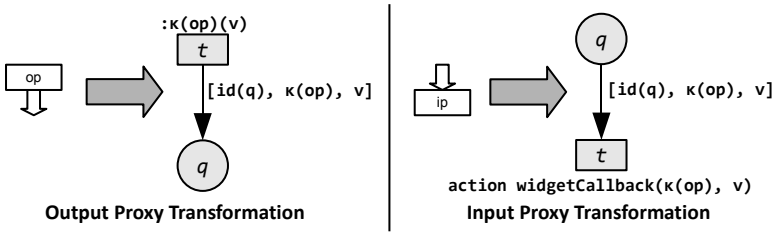


Figure 3.21.: Transformation of VFILL input and output proxies in a subnet of a reference net

Transformation of Channel Operations

In contrast to the previous transformations, the transformation of channel operations has to take into account channels defined by relation c in a FILL graph. The number of outgoing channel edges (as visual representations of channels defined by c) of an input channel operation defines the number of place-transition constructions connected to a transition that represents the channel operation itself. Furthermore, a place represents the connection to the channel operation from the perspective of interaction logic. The connections between these place-transition constructions are modeled by synchronous channels in the reference net as shown in Figure 3.22. Based on the `id` of the output channel operation, parts of the inscriptions are generated that control which transition is called to represent a given output-channel operation. First, the transformation of an input-channel operation c_I will be described in Algorithm 3.31.

Algorithm 3.31 (Transformation of Input-Channel Operation) For a given input-channel operation c_I from F, the transformation to a subnet of N will be generated.

(1) $i = 0$.

(2) v is a variable, where $\tau(v) = t(p_I)$,and $p_I \in P_I^{(c_I)}$.

(3) q_I is a place, where $q_I \notin S$, and $m_t(p_I) = q_I$.

(4) t_I is a transition, where $t_I \notin T$.

(5) u_I is a tuple variable of structure $u_I =$ `[procID, ieID,` v`]`.

(6) e_I is an edge, where $e_I \notin K$, and $k(e_I) = (p_I,\ t_I,\ u_I,\ in)$.

(7) $T := T \cup \{t_I\}$; $S := S \cup \{p_I\}$; $K := K \cup \{e_I\}$.

(8) **FOR ALL** $c_O \in C_O$ with $c(c_I) = c_O$ **DO**

- **(a)** t_i is a transition, where $t_i \notin T$, and $inc(t_i) =$ `this:channel(`$id(c_O)$`, ieID,` v`);`.
- **(b)** q_i is a place, where $q_i \notin S$.
- **(c)** u_i is a tuple variable of structure $u_i =$ `[procID, ieID,` v`]`.
- **(d)** e_{i1} is an edge, with $e_{i1} \notin K$, and $k(e_{i1}) = (q_i,\ t_i,\ u_i,\ in)$.
- **(e)** e_{i2} is an edge, with $e_{i2} \notin K$, and $k(e_{i2}) = (q_i,\ t_I,\ u_i,\ out)$.
- **(f)** $T := T \cup \{t_i\}$; $S := S \cup \{p_i\}$; $K := K \cup \{e_{i1}, e_{i2}\}$.
- **(g)** $i++$.

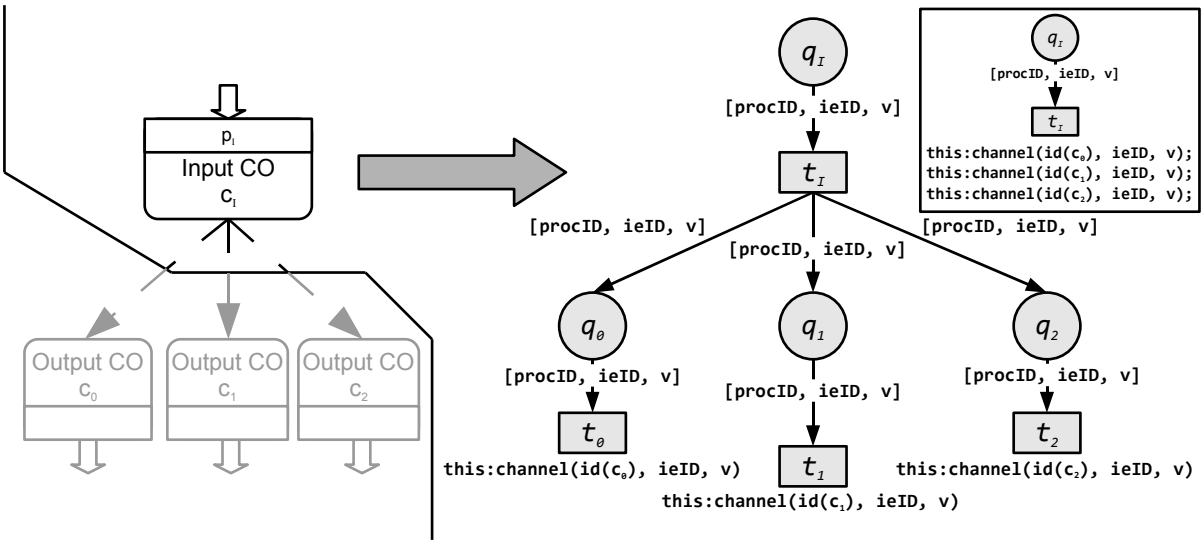


Figure 3.22.: Transformation of an input channel operation shown as a VFILL node (subnet) into a subnet of a reference net

Figure 3.22 shows the structure of an input-channel operation resulting from Algorithm 3.31. The transformed input-channel operation is connected exemplarily to three channels and thus to three different output channel operations. It can also be seen that the channels are encoded via the id of the output-channel operation that is introduced to the inscription of the transitions inscribed with the synchronous channel called `this:channel`. With this id and the unification of transitions inscription for synchronous channels, the corresponding transition representing the output-channel operation can be identified during runtime.

In the upper right of Figure 3.22, an alternative transformation can be seen. The semantic is identical, but the structure of the subnet is much simpler. The transition shown has a much more complex inscription, subsuming all downlinks to all transitions that represent output-channel operations associated with the transformed input-channel operation through channels. In spite of this apparent complexity, Algorithm 3.31 was chosen to represent channels in the reference net's structure in order to make the resulting net more understandable by a human expert and to simplify the application and creation of transformation rules to create the required net. This will be discussed in greater detail in Chapter 4.

The transformation of output-channel operations is easier than the transformation of input-channel operations. Output-channel operations only have to be transformed to a simple structure, as shown in Figure 3.23. This is because output-channel operations do not have to differentiate between different channels; they only receive calls from input-channel operations or from downlink transitions. The following pseudocode describes the conversion of an output-channel operation c_O to a reference subnet.

Algorithm 3.32 (Transformation of Output-Channel Operation) For a given output-channel operation c_O from F, the transformation to a subnet of N will be generated.

(1) v is a variable, where $\tau(v) = t(p_O^{(c_O)})$.

(2) q is a place, where $q \notin S$, and $m_s(p_O^{(c_O)}) = q$.

(3) t is a transition, where $t \notin T$, and $inc(t) =$ `:channel(`$id(c_O)$`, ieID, `v`);`.

(4) u is a tuple variable of structure $u = \texttt{[}id(p_O^{(c_O)})\texttt{, ieID, } v\texttt{]}$.

(5) e is an edge, with $e \notin K$, and $k(e) = (q,\ t,\ u,\ out)$.

(6) $T := T \cup \{t\}$; $S := S \cup \{p\}$; $K := K \cup \{e\}$.

Algorithm 3.32 is visualized in Figure 3.23, where the result of the transformation can be seen. In this way, every output-channel operation is simply transformed to a unique uplink that is indicated by the output-channel operation's id embedded in the transition's inscription.

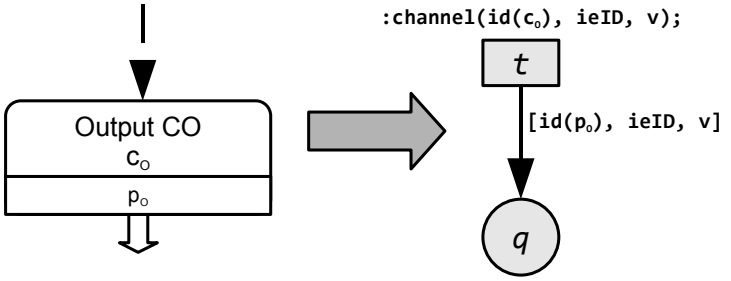


Figure 3.23.: Transformation of an output channel operation shown as a VFILL node into a subnet of a reference net

This id-based transformation has the disadvantage that output-channel operations can only be associated with one channel—a restriction that does not exist in FILL's formal definition. This means that, if n channels are connected to one output channel operation, the id-based transformation has to be changed to use ids that are associated with the channel itself, that is, to any pair (c_I, c_O) with $c(c_I) = c_O$. The resulting structure of the subnet for output-channel operations would then be similar to that of input-channel operations (except that all edges have inverted orientation, and the channel-representing transitions are inscribed with uplinks instead of downlinks).

Transformation of BPMN Nodes

In context of transformation of BPMN nodes, three different aspects have to be considered.

1. The semantics of BPMN nodes, described in Table 3.1,
2. the edge labels associated with the incoming and outgoing edges surrounding the BPMN nodes, and
3. their associated guard conditions.

The semantics of BPMN nodes undergo a structural transformation into a reference net. This means that the semantics of BPMN nodes are transformed into a specific structure of places and transitions in the resulting reference subnet. For the transformation of an OR node, the grouping labels of edges are relevant to the transformation process. For BPMN nodes without grouping, the labels are not important for the structural transformation.

Transformation of guard conditions is simple. It is based on the association of the guard conditions in FILL directly with the main transition of the transformed reference subnet representing the transformed BPMN node. The definitions of guard conditions in FILL are syntactically and

semantically similar to those in reference nets. Thus, a one-to-one conversion can be applied without the necessity of further transformation, with one exception: the reference part of a guard condition in FILL has to be split from its conditional part.

Figures 3.24 through 3.26 show the three types of BPMN nodes in FILL's visual representation and their transformation into reference nets. For the formal algorithmic transformation, an additional declaration is required as follows:

- If $b \in B$ is a BPMN node from a given FILL graph, than $E_b'^{in}$ and $E_b'^{out}$ are sets, such that

$$\begin{aligned} E_b'^{in} &= \{(p,b)|(p,b) \in E', p \in P_O \cup X_O\}, \text{ and} \\ E_b'^{out} &= \{(b,p)|(b,p) \in E', p \in P_I \cup X_I\}. \end{aligned}$$

- If $e = (s,t) \in E'$ is an edge of a FILL graph, than $src : E' \to P_O \cup X_O \cup B$ and $trg : E' \to P_I \cup X_I \cup B$ are two functions with

$$\begin{aligned} src(e) &= s, \textit{ and} \\ trg(e) &= t. \end{aligned}$$

- If $b \in B$ is a BPMN node of a FILL graph, than $m_{bt} : E_b'^{in} \to S$ and $m_{bs} : E_b'^{out} \to S$ are functions similar to m_t and m_s. Thus, S is a set of places in a reference net and not a set of system operations. m_{bt} and m_{bs} matches incoming and outgoing edges of a BPMN node b to the incoming and outgoing places representing the connector nodes in a reference net for these edges.

- $id_s : S' \to \mathcal{ID}$ 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 bijectivity of id_s, there is an inverse function id_s^{-1}.

- If $b \in \mathcal{B}$ is a guard condition, than grd and ref are functions where b_c is the conditional part of b with $grd(b) = b_c$ and b_r is the reference part of b with $ref(b) = b_r$. ref_g is a function where b_g is the group of a reference part of a guard condition with $ref_g(ref(b)) = b_g$; ref_e is a function where b_e is the referenced edge with $ref_e(ref(b)) = b_e$. For example, consider a guard condition b, with

$$b = \texttt{guard (a > 3) \& (b == true) -> g1:c;},$$

  than the functions grd, ref, ref_g and ref_e map b on

$$\begin{aligned} grd(b) &= \texttt{guard (a > 3) \& (b == true);}, \\ ref(b) &= \texttt{g1:c}, \\ ref_g(ref(b)) &= ref_g(\texttt{g1:c}) = \texttt{g1}, \text{ and} \\ ref_e(ref(b)) &= ref_e(\texttt{g1:c}) = \texttt{c}. \end{aligned}$$

For an edge label $l \in \mathcal{L}$, ref_g and ref_e map l to the edge group or to the individual edge label. For example, consider an edge label

$$l \quad = \quad \texttt{g1:d},$$

than the functions ref_g and ref_e map l on

$$\begin{aligned} ref_g(l) = ref_g(\texttt{g1:d}) &= \texttt{g1}, \text{ and} \\ ref_e(l) = ref_e(\texttt{g1:d}) &= \texttt{d}. \end{aligned}$$

- It is important in the transformation of BPMN nodes to differentiate between the name of a variable, its type and the variable itself. To do this, a function called $nam : V \to \mathcal{N}$ maps a variable in a reference net to its name as an element of set $\mathcal{N}$. All elements $n \in \mathcal{N}$ are strings and are of the same type as the edge labels of set $\mathcal{L}$ in a FILL graph.
- If $b \in B$ is a BPMN node from a FILL graph, than $E'^{in}_{g:b}$ is a subset of set E'^{in}_{b}, merging all edges that are part of the same group g. The same is true for set $E'^{out}_{h:b}$ as a subset of set E'^{out}_{b} specified by group h. Set G_b is a set of all groups used in BPMN node b.

AND Node An AND node is transformed based on the structure shown in Figure 3.24. First, the transformation of an AND node in a case of fusion will be defined in Algorithm 3.33. The types of variables have to be derived from the ports of the incoming and outgoing edges of the given AND node. Thus, $b_\oplus \in B$ is an AND node of F with n incoming edges $e_0, \ldots, e_{n-1} \in E'^{in}_{b_\oplus}$ and one outgoing edge $e_{out} \in E'^{out}_{b_\oplus}$.

Algorithm 3.33 (Transformation of a fusing AND node) For an AND node $b_\oplus$ with n incoming edges $e_0, \ldots, e_{n-1} \in E'^{in}_{b_\oplus}$ and one outgoing edge $e_{out} \in E'^{out}_{b_\oplus}$ from F, the transformation to a subnet of N will be generated.

(1) $v_0, \ldots, v_{n-1}$ are variables, where

$$\tau(v_i) = t(src(e_i)), \text{ and } nam(v_i) = l(e_i) \text{ with } i = 0, \ldots, n-1.$$

(2) v is a variable, where $\tau(v) = t(trg(e_O))$.

(3) q is a place, where $q \notin P$, and $m_{b_\oplus s}(e_{out}) = q$.

(4) t is a transition, where $t \notin T$, and $inc(t) = grd(g(b_\oplus))$.

(5) u is a tuple, where $u = \texttt{[}id_s(q)\texttt{, ieID0, }ref(g(b_\oplus))\texttt{]}$.

(6) e is an edge, where $e \notin K$, and $k(e) = (q,\ t,\ u,\ out)$.

(7) $T := T \cup \{t\}$; $S := S \cup \{q\}$; $K := K \cup \{e\}$.

(8) **FOR** $i = 0$ **TO** $n - 1$ **DO**

(a) q_i is a place, where $q_i \notin S$, and $m_{b_\oplus t}(e_i) = q_i$.

(b) u_i is a tuple, where $u_i = \texttt{[procID}i\texttt{, ieID}i\texttt{, }v_i\texttt{]}$.

(c) e_i is an edge, where $e_i \notin K$, and $k(e_i) = (q_i,\ t,\ u_i,\ in)$.

(d) $S := S \cup \{q_i\}$; $K := K \cup \{e_i\}$.

In Algorithm 3.33, the transformation of a fusing AND node has been formalized in two parts. In the first half, a transition is generated. Next, all n incoming edges are iteratively transformed into n edge/place constructions representing connections from the AND node subnet to the remaining interaction logic. The use of one transition implements the synchronization semantics of an AND node based on elements in reference nets; the transition cannot fire before all incoming places are equipped with tokens. This corresponds with the semantics of an AND node as described in Table 3.1.

Algorithm 3.34 defines the transformation of a branching AND node. In this case, $b_{\oplus} \in B$ is an AND node in F with one incoming edge $e_{in} \in E'^{in}_{b_{\oplus}}$ and n outgoing edges $e_0, \ldots, e_{n-1} \in E'^{out}_{b_{\oplus}}$.

Algorithm 3.34 (Transformation of a branching AND node) For an AND node $b_{\oplus}$ in F with 1 incoming edge $e_{in} \in E'^{in}_{b_{\oplus}}$ and n outgoing edges $e_0, \ldots, e_{n-1} \in E'^{out}_{b_{\oplus}}$, the transformation to a subnet of N is generated.

(1) $v_0, \ldots, v_{n-1}$ are variables, where

$$\tau(v_i) = t(trg(e_i)), \text{ and } nam(v_i) = l(e_i) \text{ with } i = 0, \ldots, n-1.$$

(2) v is a variable, where $\tau(v) = t(src(e_{in}))$, and $nam(v) = l(e_{in})$.

(3) q is a place, where $q \notin P$, and $m_{b_{\oplus}s}(e_{in}) = q$.

(4) t is a transition, where $t \notin T$, and $inc(t) = grd(g(b_{\oplus}))$.

(5) u is a tuple, where $u =$ `[procID, ieID,` $ref(g(b_{\oplus}))$`]`.

(6) e is an edge, where $e \notin K$ and $k(e) = (p,\ t,\ u,\ out)$.

(7) $T := T \cup \{t\}$; $S := S \cup \{q\}$; $K := K \cup \{e\}$.

(8) **FOR** $i = 0$ **TO** $n - 1$ **DO**

(a) q_i is a place, where $q_i \notin S$, and $m_{b_{\oplus}s}(e_i) = q_i$.

(b) u_i is a tuple, where $u_i =$ `[`$id_s(q_i)$`, ieID`i`,` v_i`]`.

(c) e_i is an edge, where $e_i \notin K$, and $k(e_i) = (q_i,\ t,\ u_i,\ out)$.

(d) $S := S \cup \{q_i\}$; $K := K \cup \{e_i\}$.

Algorithm 3.34 transforms a given AND node to a subnet as shown in Figure 3.24. It is similar to Algorithm 3.33 except that now n outgoing edges are activated simultaneously. To model this semantics, one transition is again used, which is activated by one incoming place/edge construction and, after it has been fired, generates n identical data objects. These data objects are sent simultaneously to all n outgoing place/edge constructions, representing the n outgoing edges of the given branching AND node.

XOR Node An XOR node is transformed on the basis of the reference net structure that can be seen in Figure 3.25. First, the transformation of a fusing XOR node will be described in Algorithm 3.35. The node $b_{\otimes} \in B$ should be a BPMN node of a given VFILL graph with n incoming edges $e_i \in E'^{in}_{b_{\otimes}}, i = 0, \ldots, n-1$ and one outgoing edge $e_{out} \in E'^{out}_{b_{\otimes}}$.

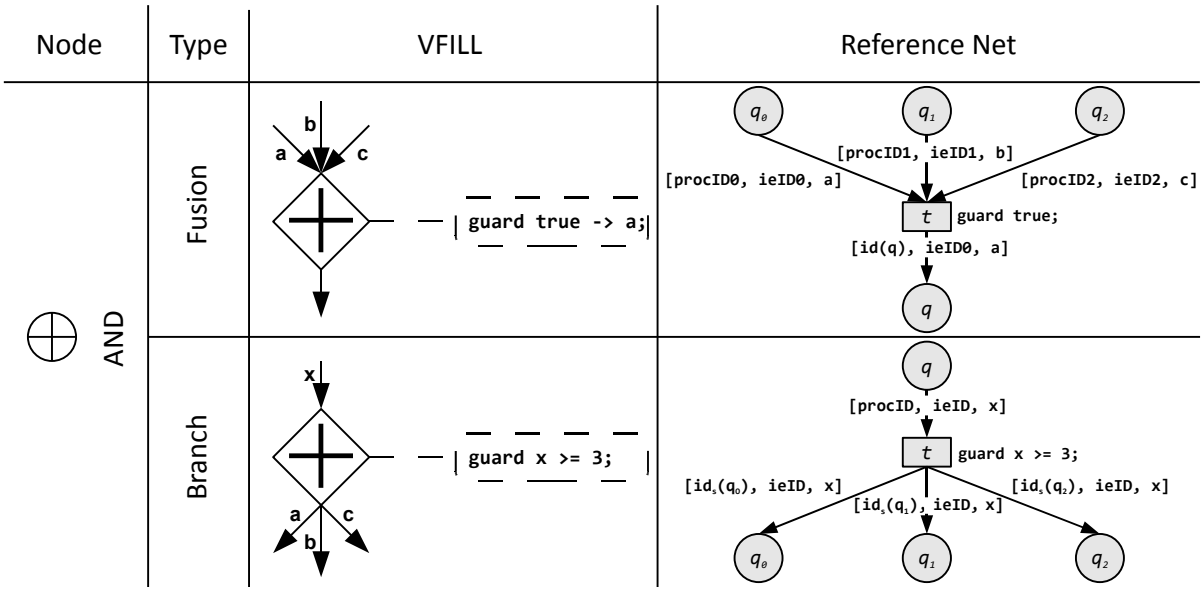


Figure 3.24.: Transformation of fusing and branching AND nodes to a reference nets

Algorithm 3.35 (Transformation of fusing XOR node) For a XOR node $b_\otimes$ in F with n incoming edges $e_i \in E'^{in}_{b_\otimes}, i = 0, \ldots, n-1$ and 1 outgoing edge $e_{out} \in E'^{out}_{b_\otimes}$, the transformation to a subnet of N is generated.

(1) $v_0, \ldots, v_{n-1}$ are variables, where

$$\tau(v_i) = t(src(e_i)), \text{ and } nam(v_i) = l(e_i) \text{ with } i = 0, \ldots, n-1.$$

(2) v is a variable, where $\tau(v) = t(src(e_{out}))$, and $nam(v) = l(e_{out})$.

(3) q is a place, where $q \notin P$, and $m_{b_\otimes s}(e_{out}) = q$.

(4) $S := S \cup \{q\}$.

(5) FOR $i = 0$ TO $n-1$ DO

(a) q_i is a place, where $q_i \notin S$, and $m_{b_\otimes t}(e_i) = q_i$.

(b) t_i is a transition, where $t_i \notin T$, and $inc(t_i) = grd(g(b_\otimes))$ with $ref(g(b_\otimes)) = l(e_i)$.

(c) u_{i1} is a tuple, where $u_{i1} =$ `[procID, ieID,` v_i`]`.

(d) u_{i2} is a tuple, where $u_{i2} =$ `[`$id_s(q_i)$`, ieID,` v_i`]`.

(e) e_{i1} is an edge, where $e_{i1} \notin K$, and $k(e_{i1}) = (q_i,\ t_i,\ u_{i1},\ in)$.

(f) e_{i2} is an edge, where $e_{i2} \notin K$, and $k(e_{i2}) = (q,\ t_i,\ u_{i2},\ out)$.

(g) $S := S \cup \{q_i\}$; $K := K \cup \{e_{i1}, e_{i2}\}$.

The semantics of an XOR node is opposite to that of an AND node. Here, during fusion, any incoming data object is sent to the outgoing edge. Thus, for every incoming edge, a place connected to a transition has to be generated to transport the incoming data object associated with that particular edge—without being affected by other incoming edges—to the

outgoing edge represented by a further place. The transformation of incoming edges is again implemented iteratively, creating the place/transition construction as described in Algorithm 3.35 and as shown in Figure 3.25.

In branching, an XOR node $b_{\otimes} \in B$ of a FILL graph will have one incoming edge $e_{in} \in E'^{in}_{b_{\otimes}}$ and n outgoing edges $e_i \in E'^{out}_{b_{\otimes}}, i = 0, \ldots, n-1$. The transformation is defined in Algorithm 3.36.

Algorithm 3.36 (Transformation of a branching XOR node) For a XOR node $b_{\otimes}$ in F with 1 incoming edge $e_{in} \in E'^{in}_{b_{\otimes}}$ and n outgoing edge $e_i \in E'^{out}_{b_{\otimes}}, i = 0, \ldots, n-1$, the transformation to a subnet of N is generated.

(1) $v_0, \ldots, v_{n-1}$ are variables, where

$$\tau(v_i) = t(src(e_i)), \text{ and } nam(v_i) = l(e_i) \text{ with } i = 0, \ldots, n-1.$$

(2) v is a variable, where $\tau(v) = t(src(e_{in}))$, and $nam(v) = l(e_{in})$.

(3) q is a place, where $q \notin P$, and $m_{b_{\otimes}t}(e_{in}) = q$.

(4) $S := S \cup \{q\}$.

(5) FOR $i = 0$ TO $n-1$ DO

(a) q_i is a place, where $q_i \notin S$, and $m_{b_{\otimes}s}(e_i) = q_i$.

(b) t_i is a transition, where $t_i \notin T$, and $inc(t_i) = grd(g(b_{\otimes}))$ with $ref(g(b_{\otimes})) = l(e_i)$.

(c) u_{i1} is a tuple, where $u_{i1} =$ `[`$id_s(q_i)$`, ieID,` v_i`]`.

(d) u_{i2} is a tuple, where $u_{i2} =$ `[procID, ieID,` v_i`]`.

(e) e_{i1} is an edge, where $e_{i1} \notin K$, and $k(e_{i1}) = (q_i,\ t_i,\ u_{i1},\ out)$.

(f) e_{i2} is an edge, where $e_{i2} \notin K$, and $k(e_{i2}) = (q,\ t_i,\ u_{i2},\ in)$.

(g) $S := S \cup \{q_i\}$; $K := K \cup \{e_{i1}, e_{i2}\}$; $T := T \cup \{t_i\}$.

In contrast to AND nodes, branching XOR nodes activate exactly one outgoing edge of the n edges. Similar to the transformation in Algorithm 3.35, Algorithm 3.36 generates one place/transition construction for every outgoing edge. If the incoming edge (here represented as one place) is activated by a data object, the guard conditions connected to the outgoing edges—and therefore to the transitions that represent them—decide which transition will fire and thus which outgoing edge will be activated.

OR Node The transformation of an OR node results in a reference net that is a mixture of an XOR and an AND node transformation because of the use of edge groups. Edges in a group behave like edges being fused or branched using an AND node; groups of edges behave like edges being fused or branched using an XOR node. Thus, if only one group exists, the semantics of an OR node is similar to that of an AND node. If any edge has its own group, the semantics of the OR node is similar to that of an XOR node. The transformation of a fusing OR node $b_{\odot} \in B$ with n incoming edges $e_i \in E'^{in}_{b_{\odot}}, i = 0, \ldots, n-1$ and one outgoing edge $e_{out} \in E'^{out}_{b_{\odot}}$ is defined in Algorithm 3.37 and can be seen in Figure 3.26.

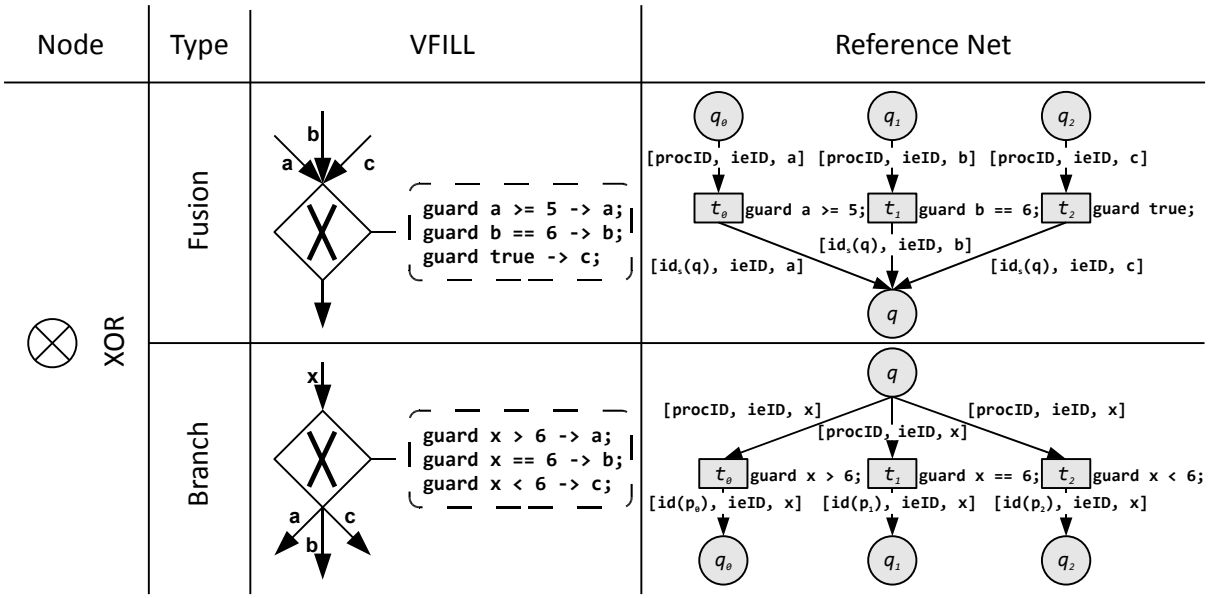


Figure 3.25.: Transformation of fusing and branching XOR nodes in reference nets

Algorithm 3.37 (Transformation of a fusing OR node) For an OR node $b_\odot$ in F with n incoming edges $e_i \in E^{\prime in}_{b_\odot}, i = 0, \ldots, n-1$ and 1 outgoing edge $e_{out} \in E^{\prime out}_{b_\odot}$, the transformation to a subnet of N is generated.

(1) $v_0, \ldots, v_{n-1}$ are variables, where

$$\tau(v_i) = t(src(e_i)), \text{ and } nam(v_i) = ref_e(l(e_i)) \text{ with } i = 0, \ldots, n-1.$$

(2) q is a place, where $q \notin P$, and $m_{b_\odot s}(e_{out}) = q$.

(3) $S := S \cup \{q\}$.

(4) **FOR ALL** $gr \in G_{b_\odot}$ **DO**

(a) t_{gr} is a transition, where

$$t_{gr} \notin T, \text{ and } inc(t_{gr}) = grd(g(b_\odot)) \text{ with } ref_g(ref(g(b_\odot))) = gr.$$

(b) v_{gr} is a variable, where $v_{gr} = v_i$ with $nam(v_i) = ref_e(ref(g(b_\odot)))$.

(c) u_{gr} is a tuple, where $u_{gr} = [id_s(q), \texttt{ieID}, v_{gr}]$.

(d) e_{gr} is an edge, where $e_{gr} \notin K$, and $k(e_{gr}) = (q, t_{gr}, u_{gr}, out)$.

(e) $T := T \cup \{t_{gr}\}$; $K := K \cup \{e_{gr}\}$.

(f) FOR ALL $e_i \in E^{\prime in}_{gr:b_\odot}$ DO

(A) q_{e_i} is a place, where $q_{e_i} \notin S$, and $m_{b_\odot t}(e_i) = q_{e_i}$.

(B) u_{e_i} is a tuple, where $u_{e_i} = [\texttt{procID}, \texttt{ieID}, v_i]$.

(C) e_{e_i} is an edge, where $e_{e_i} \notin K$, and $k(e_{e_i}) = (q_{e_i}, t_{gr}, u_{e_i}, in)$.

(D) $S := S \cup \{q_{e_i}\}$; $K := K \cup \{e_{e_i}\}$.

In Algorithm 3.37, the transformation of OR nodes to reference subnets is based on groupwise transformation, rather than edgewise transformation as is the case for the AND and XOR nodes in Algorithms 3.33 through 3.36. Nevertheless, edgewise transformation is used on the group level. Every group uses one synchronizing transition paired with places—one place for each edge in the group. The whole transformation is iteratively defined over groups and their edges. The transformation of a branching OR node $b_\odot \in B$ with n outgoing edges $e_i \in E'^{out}_{b_\odot}, i = 0, \ldots, n-1$ and one incoming edge $e_{in} \in E'^{in}_{b_\odot}$ is defined in Algorithm 3.38.

Algorithm 3.38 (Transformation of a branching OR node) For an OR node $b_\odot$ in F with 1 incoming edge $e_{in} \in E'^{in}_{b_\odot}$ and n outgoing edges $e_i \in E'^{out}_{b_\odot}, i = 0, \ldots, n-1$, the transformation to a subnet of N is generated.

(1) v is a variable, where $\tau(v) = t(src(e_{in}))$, and $nam(v) = ref_e(l(e_{in}))$.

(2) q is a place, where $q \notin P$, and $m_{b_\odot t}(e_{in}) = q$.

(3) $S := S \cup \{q\}$.

(4) **FOR ALL** $gr \in G_{b_\odot}$ **DO**

(a) t_{gr} is a transition, where

$$t_{gr} \notin T, \text{ and } inc(t_{gr}) = grd(g(b_\odot)) \text{ with } ref_g(ref(g(b_\odot))) = gr.$$

(b) u_{gr} is a tuple, where $u_{gr} =$ `[procID, ieID,` v`]`.

(c) e_{gr} is an edge, where $e_{gr} \notin K$, and $k(e_{gr}) = (p,\ t_{gr},\ u_{gr},\ in)$.

(d) $T := T \cup \{t_{gr}\}$; $K := K \cup \{e_{gr}\}$.

(e) **FOR ALL** $e_i \in E'^{in}_{gr:b_\odot}$ **DO**

(A) q_{e_i} is a place, where $q_{e_i} \notin S$, and $m_{b_\odot s}(e_i) = q_{e_i}$.

(B) u_{e_i} is a tuple, where $u_{e_i} =$ `[`$id(q_{e_i})$`, ieID,` v`]`.

(C) e_{e_i} is an edge, where $e_{e_i} \notin K$, and $k(e_{e_i}) = (q_{e_i},\ t_{gr},\ u_{e_i},\ out)$.

(D) $S := S \cup \{q_{e_i}\}$; $K := K \cup \{e_{e_i}\}$.

Algorithm 3.38 is similar to Algorithm 3.37, iterating only over groups of outgoing edges and not over groups of incoming edges.

Transformation of Data Edges

Edges are the last elements of a FILL graph to undergo transformation. It is the data edges between ports and proxies that have to be transformed. Edges representing channels are transformed in channel operations. This final transformation of data edges creates a complete reference net from all the subnets produced by the transformation of FILL nodes described above. The result is an adequate representation of a given interaction logic modeled as a FILL graph. In the final reference net, all relevant places will be connected via edge-transition constructions like the one shown in Figure 3.27. To this end, Algorithm 3.39 will transform an edge $e = (p_O, p_I) \in E'$ of F to a subnet in N connecting to the places in the subnets created by Algorithms 3.27 through 3.38.

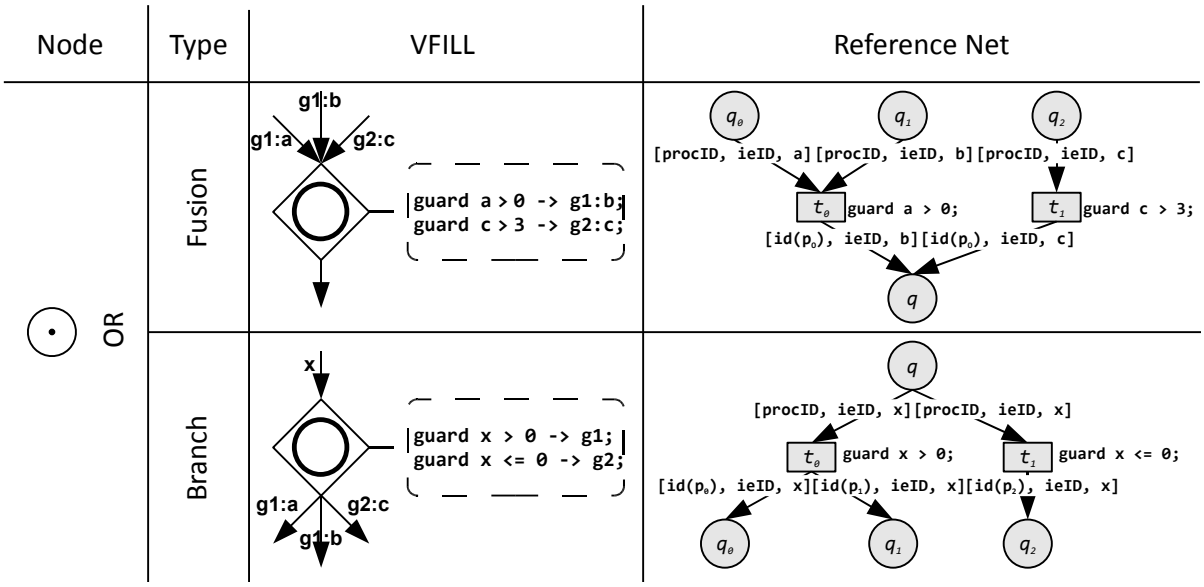


Figure 3.26.: Transformation of fusing and branching OR nodes in reference nets

Algorithm 3.39 (Transformation a data edge) For a data edge $e = (p_O, p_I) \in E'$ of F, the transformation to a subnet of N is generated.

(1) v is a variable, where $\tau(v) = t(p_O) = t(p_I)$.

(2) t is a transition, where $t \notin T$.

(3) q_O is a place, and u_O is a tuple variable, where

(a) IF $p_O \in P_O \cup X_O$

$q_O \in S$ with $m_s(p_O) = q_O$, and $u_O = [id(p_O), \texttt{ieID}, v]$.

(b) ELSE IF $p_O \in B$

$m_{p_O s}(e) = q_O$, and $u_O = [id_s(q_I), \texttt{ieID}, v]$.

(4) q_I is a place, and u_I is a tuple variable where

(a) IF $p_I \in P_I \cup X_I$

$q_I \in S$ with $m_t(p_I) = q_I$, and $u_I = [id(p_I), \texttt{ieID}, v]$.

(b) ELSE IF $p_I \in B$

$m_{p_I t}(e) = q_I$, and $u_I = [id_s(p_I), \texttt{ieID}, v]$.

(5) e_O is an edge, where $e_O \notin K$, and $k(e_O) = (q_O, t, u_O, out)$.

(6) e_I is an edge, where $e_I \notin K$, and $k(e_I) = (q_I, t, u_I, in)$.

(7) $T := T \cup \{t\}$; $K := K \cup \{e_O, e_I\}$.

In contrast to all the other transformation algorithms, Algorithm 3.39 has to identify places in N that were created before the data edge transformation in Algorithm 3.39 begins. This is because data edges are transformed into transitions connected to the places that represent ports of operations, connections to BPMN or proxy nodes after transformation.

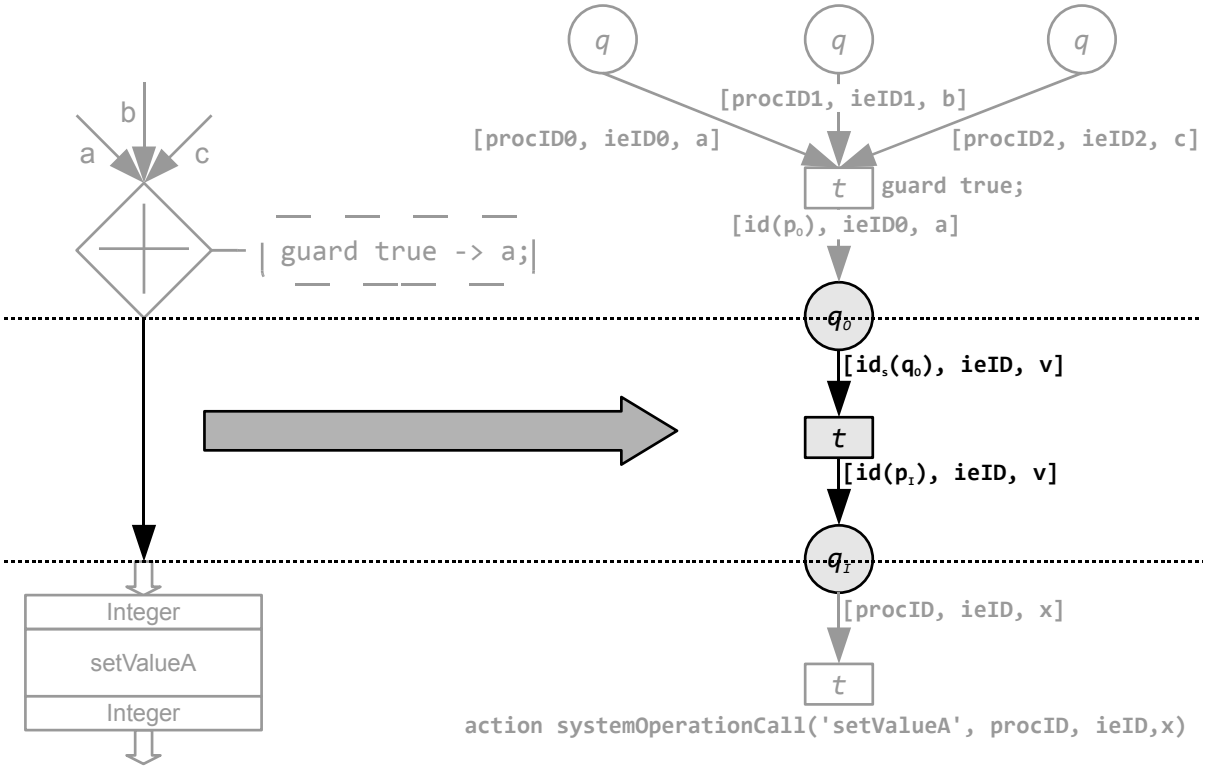


Figure 3.27.: Transformation of an edge to its representation as reference subnet

FILL Transformation Algorithm

In conclusion, the algorithms introduced above have to be embedded into an algorithm that controls the overall transformation of the FILL graph F. That is the role of Algorithm 3.40, which can be used as a super-construction to bundle the partial transformation algorithms introduced above.

Algorithm 3.40 (Transformation of a FILL graph F to a reference net N) For a given FILL graph F a representation as reference net N is generated.

(1) N is an empty reference net.

(2) FOR ALL $s \in S$ of F call Algorithm 3.27 with N and P_I and P_O of F as parameter.

(3) FOR ALL $s_{il} \in I$ of F call Algorithm 3.28 with N and P_I and P_O of F as parameter.

(4) FOR ALL $x_O \in X_O$ of F call Algorithm 3.29 with N as parameter.

(5) FOR ALL $x_I \in X_I$ of F call Algorithm 3.30 with N as parameter.

(6) FOR ALL $c_I \in C_I$ of F call Algorithm 3.31 with N and c as parameter.

(7) FOR ALL $c_O \in C_O$ of F call Algorithm 3.32 with N and c of F as parameter.

(8) FOR ALL $b \in B$ of F **DO**

- **(a) IF** b is an AND node
  - **(A) IF** b is a fusion node call Algorithm 3.33.

(B) ELSE call Algorithm 3.34.

(b) IF b is an XOR node

(A) IF b is a fusion node call Algorithm 3.35.

(B) ELSE call Algorithm 3.36.

(c) IF b is an OR node

(A) IF b is a fusion node call Algorithm 3.37.

(B) ELSE call Algorithm 3.38.

(9) FOR ALL $e \in E'$ of F call Algorithm 3.39 with N as a parameter.

Example of a Transformed FILL Graph

To conclude the current section, a transformed FILL graph will be presented in Renew's visualization of reference nets. Besides providing a simulator for reference nets, Renew also offers a visual editor for modeling reference nets interactively or importing and exporting various types of serializations. For the simulation, Renew uses a data structure called a *shadow net system*, which represents a reference net in a form that is suitable for simulation without style information like the position or size of transitions or places. This information is added as a *net drawing* to the shadow net as visualized in Figure 3.28.

Figure 3.28 shows part of the interaction logic of a user interface. Its corresponding physical representation can be seen in Figure 3.29, indicating parts that are associated with the partial interaction logic in Figure 3.28. The interaction logic in Figure 3.28 describes the processing of press events of the key labeled 'SV2' and the status of a lamp directly above it. Pressing this key results in an open or close action of steam valve 2 in a simulation of a nuclear power plant (see Section 7 below). The decision to open or close the valve is modeled into the interaction logic. As can be seen on the left in Figure 3.28, a press event is sent to a system operation returning the current value set to SV2, which is further processed using an XOR node. Depending on its guard condition, if the current value is `false`, edge `a` will be activated. Otherwise, edge `b` will be activated. If edge `a` is activated, a Boolean value `true` will be generated and sent to the system operation `setSV2Status`. If edge `b` is activated, a Boolean value `false` will be generated, and the valve will be closed. Figure 3.28 (right side) depicts the interaction logic for controlling the status lamp. Here, an interaction-logic operation called `ticker` sends simple data objects of type Object to the interaction process. This data object triggers the `getSV2Status` system operation, which returns the current status of SV2. The Boolean value resulting from this operation is directly sent to the input proxy representing the status lamp on physical representation layer.

In addition to the interaction logic described as a FILL graph, Figure 3.28 shows its transformation to a reference net. Here, the various processes involved in the transformation are associated with the original elements in the FILL graphs. Thick lines connect the elements in the FILL graph and the reference net, which are indicated by overlays with different types of border; the dashed boxes indicate parts of the left interaction process (associated to the key), and the dotted ones parts in the reference net resulting from transformation of the right interaction process (associated to the lamp). The transformation itself is not further described because the visual representation is simple, and the process can be easily derived from the visual

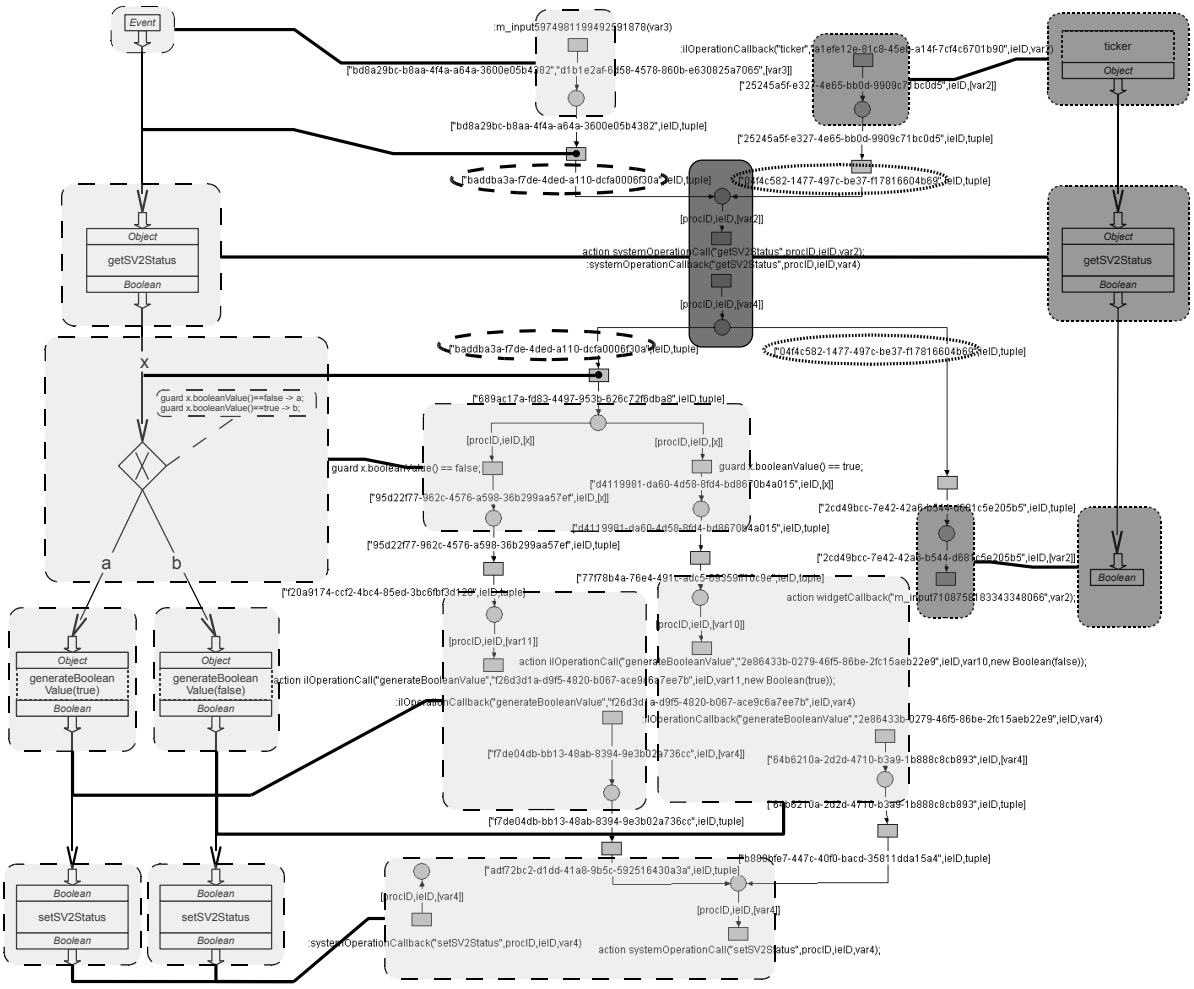


Figure 3.28.: Example of partial interaction logic and its corresponding transformation to a reference net

representation of transformation algorithms introduced above. Nevertheless, one special feature of the implementation of the transformation algorithm can be seen in Figure 3.28. Transformations of system operations are fused to only one subnet in interaction logic. Here, the system operation `getSV2Status` has been used by the interaction processes for both the key and the lamp. This reduces the number of elements in the reference net, thus enhancing performance. Still, the interaction processes have to be distinguished, which is accomplished by using Process ids (`procID`). As can be seen, the process ids specified in the tuple associated with the incoming edges are different, and they only match specific tuples associated with the outgoing edges.

The next section will introduce some approaches to modeling the physical representation of a user interface formally. In addition to some basic approaches based on XML description language, an ongoing approach to modeling interaction elements will be introduced.

3.5. Extension of Interaction Logic Modeling

The presented architecture (cf. Figure 3.2) is closely related to other well-known concepts, such as the *Seeheim Model*. This layered model was published in context of a workshop in 1983 [217]. As Figure 3.30 shows on the left, the Seeheim model differentiates between three layers: (a) the

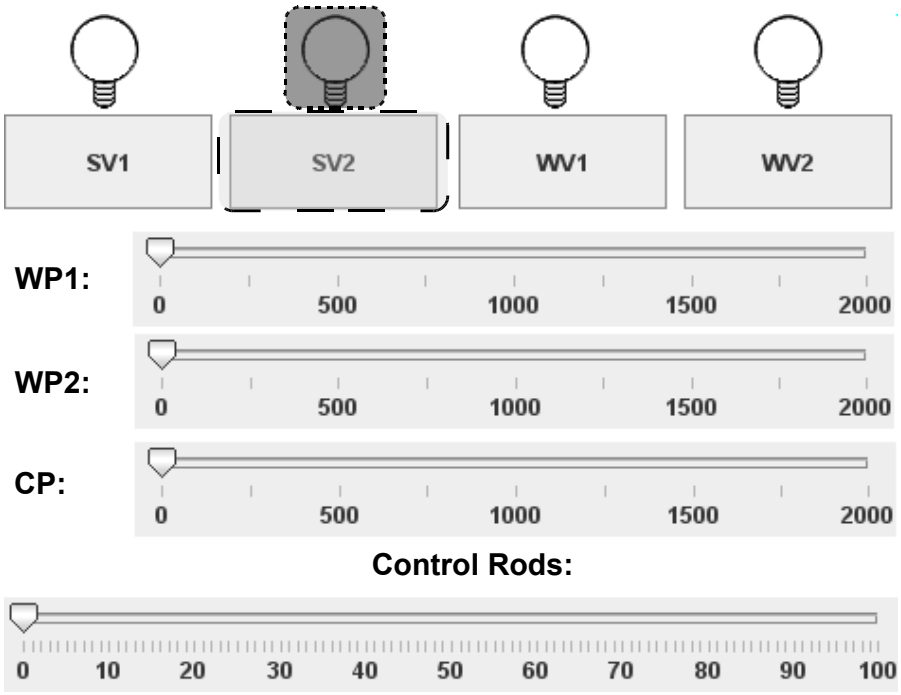


Figure 3.29.: Physical representation of a user interface showing elements associated with the interaction logic presented in Figure 3.28

presentation layer, which is basically equivalent to the physical representation of a formal user interface, (b) the dialog control component, which has the same task as the interaction logic, and (c) the application interface model, which is more or less equivalent to the system interface as introduced in this work. Nevertheless, an enclosed formal modeling approach like the one introduced in this work does not exist.

As further research has shown, this kind of rough two- or three-layer model is not fine-grained enough for certain purposes. Thus, models like the ARCH model [300] (as is shown in Figure 3.30 on the right) have been further developed to define a more detailed structure of user interface models, thus offering a better approach for describing user interfaces in greater detail. Here, the component specifying dialog has been extracted from the technology-dependent perspective, where it was represented in the Seeheim model. Using translation components such

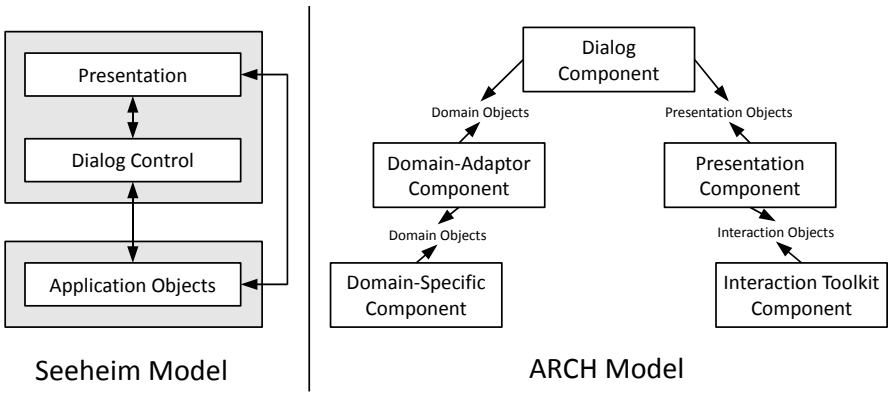


Figure 3.30.: Seeheim and ARCH models as architectural approaches to user interface modeling

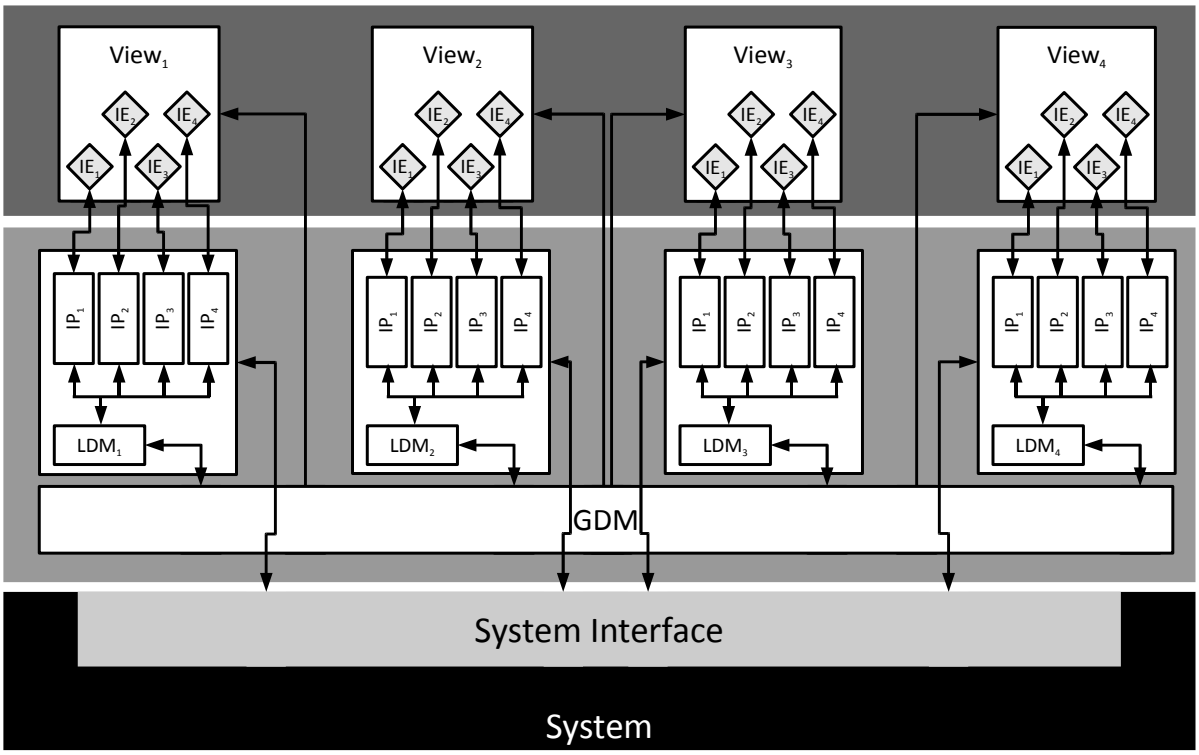


Figure 3.31.: Extended user interface model considering local (LDM) and global dialog models (GDM), differentiating communication between the user interface and the system from its dialog model

as the domain-adapter component to translate task-level sequences from the dialog component to the domain-specific component (denoted as system in this thesis) makes it possible for the dialog model to be exchangeable without necessarily exchanging the domain-specific component or, on the other side, the interaction toolkit component, which is equivalent to the physical representation. Still, this model makes the domain-specific component and the interaction toolkit component as exchangeable as the dialog component. Based on these findings, it should be of interest to extend the basic model proposed in this thesis to one offering finer grained and exchangeable components, identifying meta-models in the sense of dialog model components, and, furthermore, to discuss to what extent these components can be managed using VFILL and what role an extension to hybrid models can play.

Component-based User Interface Modeling with FILL

A first step towards component-based modeling of a user interface was made in the context of FILL by the called *interaction processes* introduced above, which can be understood as a sub-graph of the entire FILL-based interaction logic, which is exclusively associated with one interaction element of the physical representation. The interrelation between different interaction elements is part of the different interaction processes that are connected with each other by channels. This kind of modeling approach of dialog aspects in the interaction logic produces a fixed-dialog model entangled in the interaction logic, which was identified as a problematic aspect in earlier research (cf. the Seeheim and ARCH models discussion above). Thus, in future work, the basic approach of component-based modeling using the concept of

interaction processes should be extended by a local dialog model describing the interrelation between the interaction elements using FILL. Here, a possible approach could be to represent interaction processes by interaction-logic operation nodes in which input and output ports represent input and output channel operations in the interaction process, so that it is possible to send specific information to the dialog model from the interaction process and vice versa. In that local dialog model, dependencies and interrelations can be modeled on the basis of data sent by the interaction processes. This makes the interaction processes and the dialog models exchangeable and modularizes interaction logic, as can be seen in Figure 3.31. Here, the dialog model is indicated as local dialog model or LDM.

The term *local* refers to the fact that user interfaces modeled in the sense of a general two-layered concept only allow one physical representation to be modeled; that is, it is not possible to model multiple views that can be changed depending on a global dialog model (GDM), as is the case in today's user interfaces. For instance, the user interface changes if new data has been loaded that needs a different visualization and a different way to handle editing or changes in presentation. In future work, the user interface model should be extended by a GDM. This GDM handles physical components or views of the physical representation of a user interface, such as a menu, a central component, or the like. Again, information from the GDM can be passed to and from the various LDMs of the elementary user-interface components via channel operations. LDMs could also be represented as interaction-logic operations on the level of the GDM, which also makes GDMs be modeled by FILL. The whole resulting architectural structure can be seen in Figure 3.31, where the physical representation is split into several views. Each view is connected to an LDM, and the GDM handles the interaction of the various views.

The whole extension is also supported by reference nets, which are important because dialog models are also modeled using FILL. Thus, a transformed FILL-based dialog model can also be represented as reference nets. Using the reference mechanism, it is possible to modularize interaction logic as described above using various nets instead of only one. The transformation of channel operations will support a successful integration of the component-based architecture into the existing formal approach.

Hybrid User Interface Models

Still, it would sometimes be helpful to integrate other kinds of formal models into FILL in order to extend the possible range of modeling to other kinds of control structures and information-generating models, as described, for instance, in [287]. Here, models of errors in interaction implemented as deterministic finite automatons are integrated into reference net-based interaction logic to identify interaction errors during runtime and trigger relevant counteractions to prevent fatal system errors. Based on FILL and its transformation to reference nets, it is possible to integrate further modeling approaches to interaction logic. The example mentioned above of integrating a deterministic finite automaton into reference net-based interaction logic is only one specific example. Based on the concept as described in [287], adding third-party models to interaction logic using the referencing mechanism of reference nets also makes it possible to integrate other kinds of formal models (probabilistic or mathematical models such as differential equations) to the FILL-based modeling approach. These models can be bound to interaction-logic operations and also, in this way, transformed into reference nets as described above. The resulting reference net relates only to the specified model, which has also been simulated in a specific simulation implementation during runtime. Therefore, the presented ap-

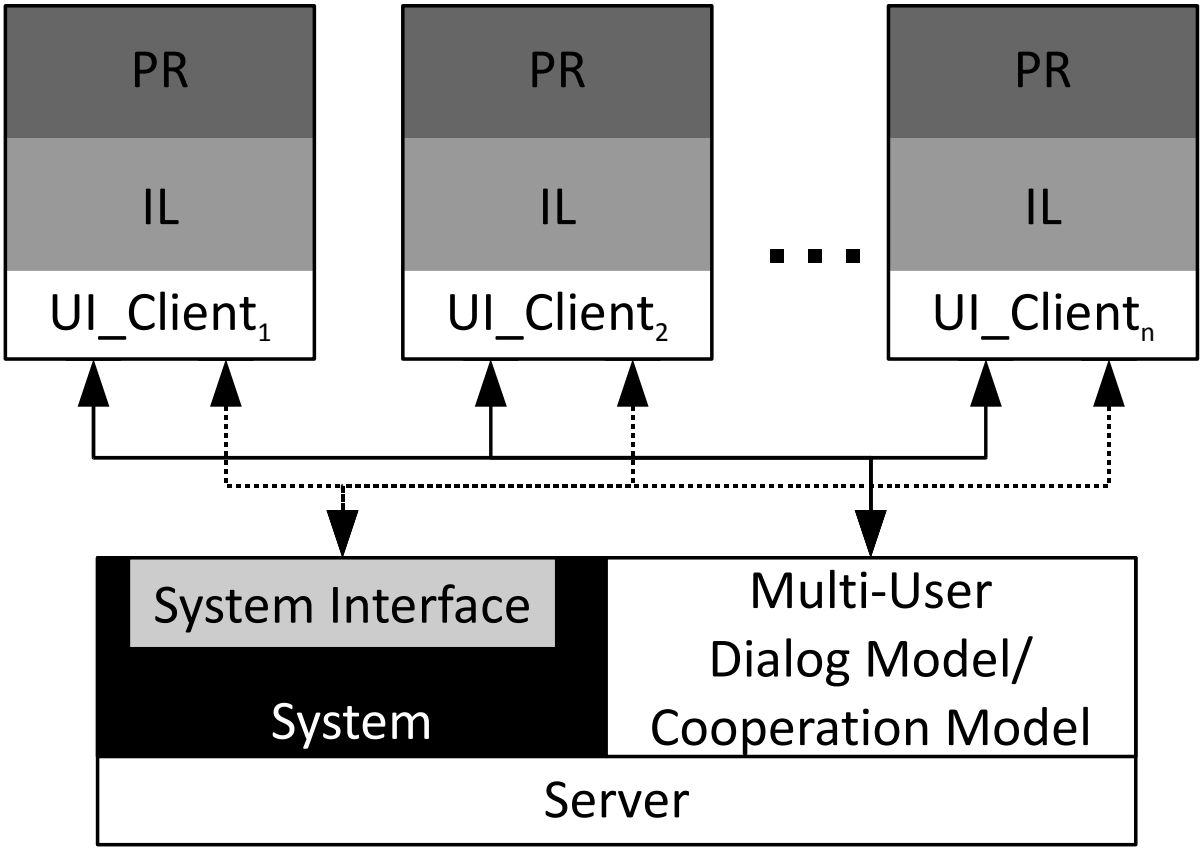


Figure 3.32.: Multi-user interface architecture to be modeled with FILL based on a client-server architecture

proach can also be extended through further modeling approaches and, thus, is not only capable of being modeled in a component-based fashion but also extensible for subsequent use of other formal approaches without it being necessary to instigate a possibly erroneous transformation to FILL.

Multi-User Interfaces

The last area of future work concerning interaction-logic modeling in this architectural context is the modeling of multi-user interfaces. Here, multi-user interfaces are user interfaces that are operated by more than one user in a cooperative or asynchronous fashion. Different scenarios are possible. Probably the most common of these is the one-interface-multi-user scenario, where one physical user interface exists but several users operate it. Another possible scenario is one in which multiple interfaces are operated by multiple users. A case in point is cooperative systems implemented on mobile devices. Here, less research has been done, especially from the perspective of user-interface modeling and creation. Most work has focused on the problem of creating user interfaces for devices with small monitors. FILL could serve as a modeling language for multi-user dialog models, as can be seen in Figure 3.32. Here, the multi-user interface is based on a client-server architecture, where the multi-user dialog-model runs on the server and, simultaneously, on the n user interfaces; that is, for every user interface employed, one runs on a client. These single-user interfaces are therefore modeled as described above based on the formal component-based modeling approach.

3.6. Formal Modeling of Physical Representation

Section 3.2 introduced the physical representation as a tuple (I, P, C, p, c), where

- I is a set of interaction elements representing the elementary items of a physical representation of a user interface,
- P is a set of parameters defining specific attributes of certain interaction elements, where $p : I \to P^*$ is a total function mapping every interaction element in I to a set of parameters specifying their attributes, such as their outward appearance, and
- C is a set of classes of interaction elements, where $c : I \to C$ is a total function that maps every interaction element of a given physical representation to a class of interaction elements.

This simple and basic definition of a physical representation can be mapped to many publicized standards for describing physical representations and user interfaces modeling languages, which also include interaction logic elements in the description.

Past research has shown XML to be a suitable approach for standardized serialization of complexly structured data. Especially in the modeling of user interfaces, XML has had a great impact on the standardization of data formats for parsing, reading, and publishing interactive content. One common example is XHTML [274], which is based on the SGML language standard published by ISO [119]. A specific markup language for describing user interfaces is XUL [221], which was developed as part of the Mozilla project[4] and is now the standard language for modeling user interfaces in the Firefox Browser [89]. The logic and the connection to the system is implemented using JavaScript [92, 118], which itself is the standard in programming dynamic content for the World Wide Web in AJAX [51], as well as other technologies, including Flash [97] and ActionScript [237]. Still, XUL is application-dependent in the sense that a user interface described in XUL can only be rendered and used in the Firefox Browser or in other Mozilla framework-based implementations. There are some open-source projects for open XUL for other platforms, like Java Swing, but they are still closely related to Mozilla's framework.

Another example that is independent of certain implementations is UsiXML [273]. UsiXML is a markup language for modeling user interfaces embedded in a huge framework for user interface development unifying various XML-based languages, like *TransformiXML*, *IdealXML*, and *GrafiXML*. This framework implements workflows for developing user interfaces in different stages, from a model of *task and concept*, to an *abstract user interface model*, to a *concrete user interface*, and, finally, to a *final user interface*. Various tools supporting this workflow were developed mainly to support the transformation of modeled user interfaces to other contexts with a minimum of effort[5].

A further example is UIML, which is a markup language developed and standardized by the OASIS standardization committee [190]. UIML was first developed by Phanouriou [218] at the Center of Human-Computer Interaction at the Virginia Polytechnic Institute and State University in 2000. It was developed mainly as a platform-independent markup language for modeling user interfaces involving the style, structure, content, and behavior of a user interface. These terms are not exactly the same as those used above. Here, the *structure* of a user interface describes how elements of the user interface are positioned. *Style* defines style-specific

[4] http://www.mozilla.org
[5] http://www.usixml.org/index.php?mod=news

aspects, such as color and font. The *content* of a UIML description defines the content of the user interface's elements, mainly text, images, and so forth. *Behavior* defines how the user interface changes its outward appearance in relation to different system states and the elements' interrelation. *Logic* defines the connection with a given data source, and *presentation* specifies the parameters for any given device.

These description languages focus primarily on describing the physical representation of a user interface. In some cases, they also include basic approaches to describing the behavior of a user interface concerning events, etc. Still, there is no language, such as FILL, embedded to describe complex data processing and the interrelation of interaction elements based on the three-layered architecture described above. Also, the necessary specifications for connections between interaction elements and interaction logic are not provided as they are in the explicit connection between interaction logic and system interface shown above.

To address this lack, a proprietary XML-based format was implemented that provides the specific information needed to combine the physical representation and the interaction logic modeled as a reference net. Also, the flexibility of a user interface is also needed for reconfiguration. The proprietary format, which is described in Section 5, could be transformed to UIML or to another XML-based markup language to describe the physical representation. This transformation, when implemented in a format such as an XSLT schema [275], offers a standardized format for the distribution of a user interface.

The introduction of multiple views extends to the basic set-based definition of physical representation. This extension would offer dialog-based modeling as discussed above in the context of GDMs and multi-user interfaces. Therefore, the physical representation $PR = (I, P, C, p, c)$ will be extended as given in the following definition.

> **Definition 3.41 (Multi-View Physical Representation)** *A multi-view physical representation is a tuple $MPR = (V, I, P, C, p, c)$ where I, P, C, p, c are equally defined as in Definition 3.14 and $V \subseteq \mathcal{P}(\mathcal{I})$, where $\mathcal{P}(\mathcal{I})$ is the power set of I.* ◇

Definition 3.41 presents a simple extension to the basic approach as introduced in Definition 3.14 but makes the modeling of multiple-view user interfaces possible. Therefore, the GDM can handle the behavior of the different views and also define, for instance, a menu-based change of views.

Modeling Interaction Elements

These XML-based approaches are suitable for the standardized modeling of user interfaces, for instance, for highly exchangeable interfaces in mobile environments or for cross-platform use. Thus, modeling the physical representation of a user interface is only possible in a restricted way when using XML-based description languages. To offer a broader solution to this problem of the restricted and static set of interaction elements, a concept for interactive and visual modeling of interaction elements was developed in cooperation with Sema Kanat [131] and Alexander Emeljanov [81]. The main goal of using formal methods to model interaction elements was to combine the input information mainly associated with mouse events on the part of the interaction element with a state machine. The state machine, which is described as a reference net, defines the behavior of the interaction elements in response to input events or values.

This approach to the modeling and implementation of interaction elements combines a visual modeling step with a connection to an automaton-based model of the changes applied to the

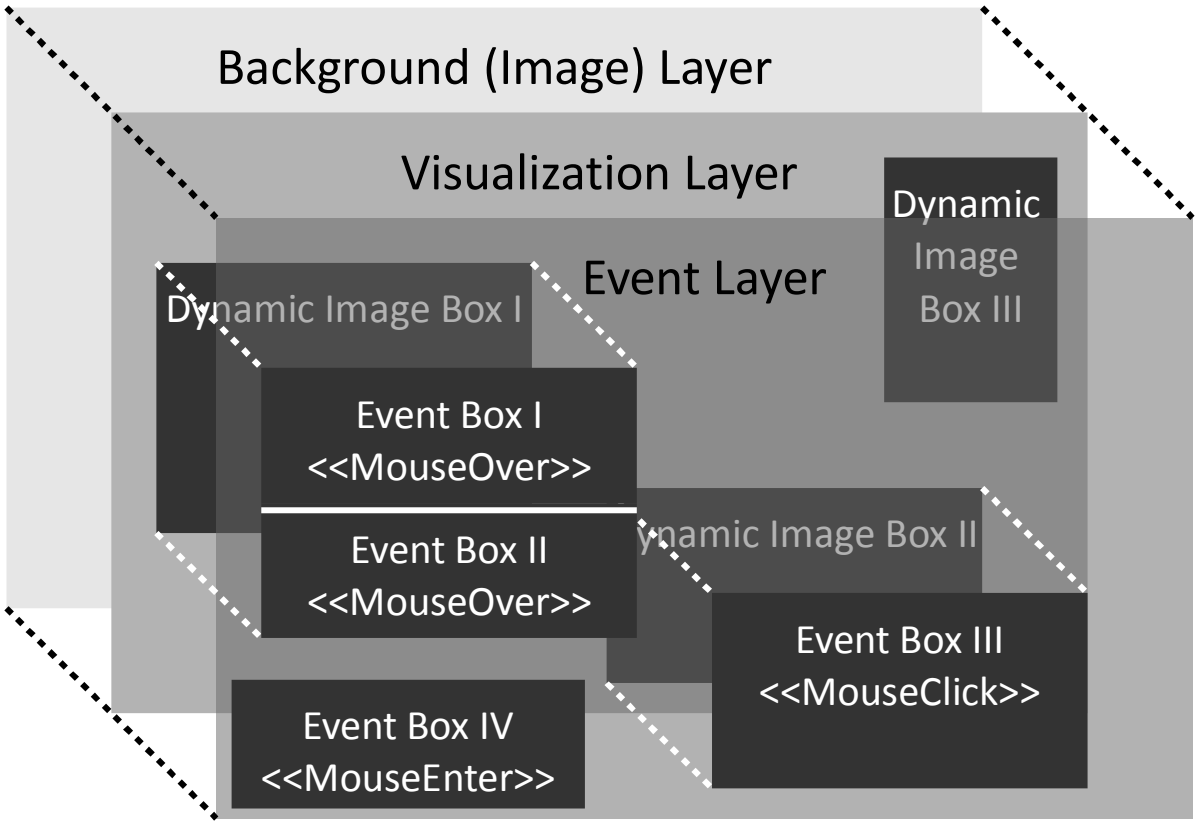


Figure 3.33.: Architectural concept for modeling and creating complex interaction elements using visual editors and reference nets

outward appearance of the modeled interaction element. To this end, interaction elements are modeled physically based on a layered architecture, as shown in Figure 3.33. An interaction element consists of three layers: (a) the background layer, showing a background image and defining the outer bound of the interaction element; (b) the visualization layer, which is composed of *Dynamic Image Boxes (DIB)*; and (c) the event layer, which is also composed of *Event Boxes (EB)*. DIBs are frames showing a set of images one at a time that can be changed in accordance with a state change modeled by the interaction element's logic (implemented as a reference net or state machine). EBs are areas that react to mouse events of various types, which can be specified individually for every EB. In Figure 3.33, the type of event is indicated textually.

Figure 3.34 depicts an example of an interaction element as a layered implementation, a set of images for the DIB *DIB_a*, with its logic modeled as reference net. This interaction element is a combination of input and output elements representing a machine status that distinguishes between *normal*, *critical*, and *accident* system states. As can be derived from its logic, in a normal system state, the interaction element shows a key-like image in green. In other states, the green image is replaced by an orange or a red image. As shown by the uplinks at the `status` transitions, the status has to be sent from the system to the interaction element. The same is true for the processing of events. The visualization engine that renders the interaction element and captures the mouse events has to handle the triggering of the uplink-event transition (seen on the right side of the reference net in Figure 3.34). The system information can also be processed by interaction logic, which is also modeled as a reference net. The mouse event

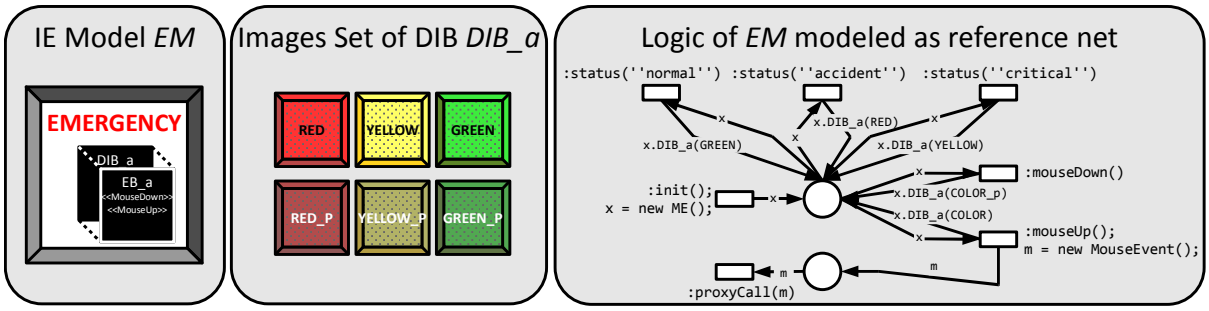


Figure 3.34.: Example of a created interaction element using a layered architecture shown in Figure 3.33 with its logic modeled as a reference net

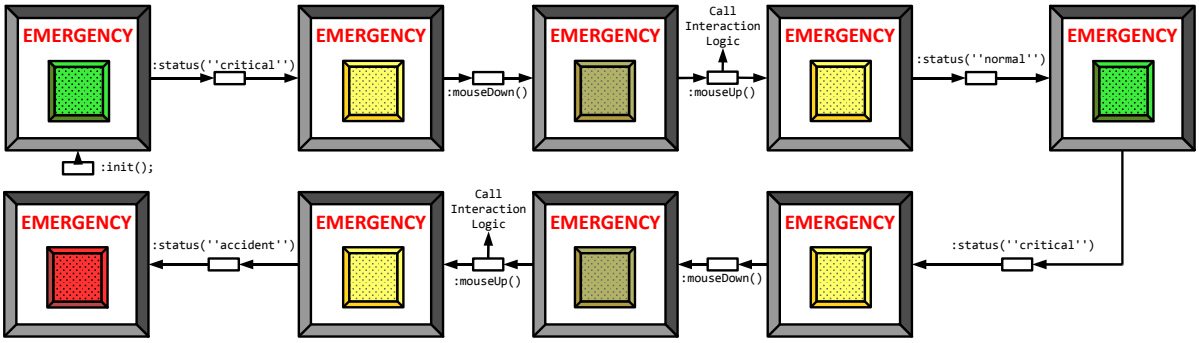


Figure 3.35.: A possible behavior of the modeled interaction element in Figure 3.34 showing the inputs resulting from the system or the user

that is sent to the interaction element will also be processed in the interaction logic later on. Therefore, a downlink transition was added to the interaction element's reference net (labeled `:proxyCall(m)`, indicating the connection to a proxy in interaction logic). Figure 3.35 shows the modeled interaction element during runtime.

This approach to modeling interaction elements for use in a physical representation is a consistent continuation of the formalization of user interfaces. This image- and box-based approach is currently under investigation, as are the various ways it can be extended. These include offering more complex types of boxes (circles, polygons etc.), image types (gif. animation etc.), and more complex and more flexible visualization approaches (canvases, 3D rendering areas etc.). Modeling parts of the physical representation of an interface in this way makes it possible to formalize not only the behavior of the user interface as interaction logic, but also to introduce the 'logic' of interaction elements and their behavior in response to input from the interaction logic or from the user, such as key or mouse events.

3.7. Conclusion

This section discussed the need and opportunities for formal modeling of interaction logic in modeling user interfaces. The first step was to introduce a nomenclature that distinguishes the terms used here from their general use in the literature so as to avoid misunderstandings and erroneous interpretations. Next, a new approach to a three-layered architecture for modeling user interfaces was introduced that differentiates between (a) the physical representation, (b) its interaction logic, and (c) the system interface that represents the observable and manipulable

parts of the system to be controlled. Then, a new formal language called FILL was described along with its visual representation. FILL was developed mainly to model the processing of data triggered by the human user of a physical representation or data sent from the system. The resulting graph-based language lacked formally defined semantics. Therefore, the formal transformation of FILL graphs to reference nets was introduced and algorithmically described. The use of reference nets has shown a number of advantages concerning the requirements that result from the formalization of interaction logic. Thus, some requirements were also generated by a later implementation for modeling, simulation, and analysis of formal interaction logic.

In addition to the formal modeling of interaction logic based on FILL and its transformation to reference nets, a short introduction to possible approaches to formal modeling of the physical representation of a user interface was discussed. The formal modeling of a physical representation is not in the focus of this work, but is still an important aspect of modeling formal user interfaces. Some XML-based description languages were briefly introduced as possible candidates for formal modeling. Finally, an approach developed in cooperation with Kanat and Emeljanov was described, which is currently being investigated.

After introducing the formal modeling of interaction logic, the next step is to describe an approach to reconfiguring interaction logic. In this context, reconfiguration can be understood as a tool for adapting user interfaces on the logic level. Formal reconfiguration means that adaption becomes verifiable in various senses. Furthermore, formal reconfiguration makes it possible to stay in one formalism—here, the use of reference nets. Another motivation for using formal concepts for reconfiguration is implementation. Using a formal approach compatible to reference nets makes implementation in a higher programming language easy without losing expressiveness, which might occur if a complex and informal approach for reconfiguration has to be transformed to a formal programming code. Therefore, the next chapter introduces formal reconfiguration, as well as a formal redesign for adapting physical representations. It will also deal with graph rewriting concepts and their customization for use in reference nets and formal interaction logic.

4. Reconfiguration and Redesign

Reconfiguration and redesign are two terms that are often used synonymously. Therefore, this chapter starts by defining these terms more clearly and bringing them into the context of formal modeling of user interfaces (Section 4.1) in the same way as was done for nomenclature in the previous chapter. Formal reconfiguration of interaction logic as one part of the adaption of user interfaces will be further introduced and discussed. Then, approaches to the reconfiguration of user interfaces in different application scenarios will be investigated, with a particular focus on various reconfiguration operations (Section 4.2). This section will be followed by an introduction to formal redesign techniques for adapting the physical representation of a user interface (Section 4.3). The chapter will conclude with a summary of the concepts that have been introduced and give a short preview of the next chapter (Section 4.4).

4.1. Nomenclature

This section will give an overview of the nomenclature used as an extension of the concepts presented in Section 3.1. The three-layered architecture for modeling user interfaces using formal languages presented there motivates a closer look at the definition of the terms used to describe and define formal reconfiguration and redesign. The partitioning of a user interface into its physical representation and its interaction logic necessitates defining terms relevant to adaption. To do this, adaption will be split into two different processes: (a) *reconfiguration*, which denotes the adaption of interaction logic and (b) *redesign*, which refers to the adaption of the physical representation of the user interface. The combination of reconfiguration and redesign can be called adaption of a user interface independently from the system that applies these kind of changes.

> **Definition 4.1 (Reconfiguration)** Reconfiguration *of a user interface is the adaption of interaction logic by applying given adaption rules. Formal reconfiguration applied to formal interaction logic is the application of formally defined transformation rules.* ◇

Changes and reconfigurations of interaction logic (formal or non-formal) are often paired with the adaption of the physical representation of a user interface.

> **Definition 4.2 (Redesign)** Redesign *of a user interface is the adaption of its physical representation by applying given adaption rules.* ◇

Adaption rules for redesigning a user interface as defined in Definition 4.2 influence the parameters of the physical representation's interaction elements, for instance, size, background color, and position.

Because of their close relationship, the terms *reconfiguration* and *redesign* sometimes cannot be clearly differentiated. Reconfiguration as applied to interaction logic often involves the

redesign of the physical representation. For instance, deleting an interaction process from an interaction logic also results in deleting its associated interaction elements in the physical representation. Thus, in the following sections, the term *reconfiguration* will sometimes be used to refer to both the reconfiguration of an interaction logic and the redesign of a physical representation.

In the following sections, reconfiguration and redesign will be introduced on both a formal and an informal basis. In particular, the reconfiguration of interaction logic will be investigated and discussed in detail by applying formal graph transformation systems adapted to reference nets. Furthermore, in the context of a short look at implementation that will be introduced in Chapter 5, a graph transformation system based on XML will be presented.

4.2. Formal Reconfiguration

Chapter 3 introduced a formal approach to modeling interaction logic based on a formally defined, graph-based language called FILL, its visual representation called VFILL, and its transformation to reference nets providing formal semantics. For reconfiguring formal interaction logic, a formalism has to be identified that (a) can define reconfiguration rules in relation to reference nets or be extended to apply to reference nets and (b) that is a well-known approach for adapting graph-based structures.

In the context of the later requirement, the choice of graph transformation systems seems to be the correct one. The research area of graph transformation systems refers to a variety of approaches that change graphical structures by applying rules in a certain way. There are two main approaches to graph transformation systems: (a) graph grammars and (b) graph rewriting systems [149].

Both graph grammars and graph rewriting systems are based on the concept of rules. Still, the approaches differ in their objectives: Graph grammars generate a net by using a set of transformation rules and an initial graph, while graph rewriting systems instead apply rules to an existing net to change its structure without seeking to produce or reproduce the graph. Thus, graph grammars produce graphs, whereas rewriting systems replace, delete, or add parts of an existing net. In formal reconfiguration, a graph rewriting approach makes the most sense. A standard scenario for reconfiguring interaction logic is that an initial interaction logic, such as a reference net, has to be adapted in a certain way. Thus, it is contradictory to use an approach like graph grammars, which creates a completely new graph any time a rule is applied to the existing interaction logic. A further argument against the use of graph grammars is that the initial interaction logic would have to be given as a set of generating rules, which is contrary to the approach detailed above (cf. Section 3.4), which describes FILL's transformation to reference nets and not to a set of generating rules. For these reasons, the following explanations introduce the use of graph rewriting systems applied to reference nets as the formal basis for reconfiguration.

Petri Net Rewriting

Ehrig and Nagl [70, 178] gave an overview of graph transformation systems. Newer works, like Heckel [107], provide a short introduction to the application of graph grammars. The *Handbook of Graph Grammars and Computing by Graph Transformation* presents a broad selection of various approaches to the use of graph grammars and graph transformation techniques and

their use in various application scenarios, like software engineering or pictures [72]. Schür and Westfechtel [246] identify the following groups of graph rewriting systems:

Logic Oriented This approach defines rules and their context of application using predicate logic. This powerful approach is not widespread because of its complex implementation.

Set Theory Rules for rewriting graphs can be defined on the basis of set theory. This approach is highly flexible and can be applied easily to an application scenario. Still, with this approach possible irregularities can appear in the adapted graph. Furthermore, problems arise in applying mathematical verification and validation concepts to a specialized set-oriented rewriting system.

Category Theory Graphs and graph morphisms that are defined on the basis of these graphs can be defined as a mathematical category. Rules for graph rewriting are then defined as morphisms in that category and often modeled as pushouts. A pushout is a category theory concept that defines a colimit of a diagram of two morphisms (explained in more detail below). Two main approaches are based on pushouts: (a) the single-pushout approach and (b) the double-pushout approach. In the single-pushout approach, one pushout is defined for deleting and adding graph components for the graph to be rewritten. In the double-pushout approach, one pushout is defined for deleting and a second for adding components to a given graph. The approach based on pushouts has limited expressiveness compared with the other two approaches. Nevertheless, this approach is simple enough to handle for formal verification without dealing with the sorts of problems that can occur in set theory-based approaches, which require great mathematical effort. Using pushouts is still powerful enough for the reconfiguration of reference nets and, in general, interaction logic, as will be seen below.

Taking all this into consideration, a category theory-based approach is a good choice, reducing complexity to a minimum for implementation, without losing the ability to apply mathematical verification methods. It is also a widespread approach to formalizing graph rewriting systems and is supported by a broad formal basis and a wide range of publicized research.

Next, graph rewriting has to be deployed to Petri nets and then to reference nets in order to formally implement the reconfiguration of interaction logic. Graph rewriting applied to Petri nets has been subject of investigation in several publications. Wileden [295] describes the use of graph rewriting for modeling Petri net semantics as used in the work of Kummer [149]. Wilke [296] gives an overview of the use of graph rewriting concepts in Petri net-based formalisms. In this work, works by Ehrig et al. [74, 77] will be applied to reference nets. Ehrig uses the double-pushout approach for the transformation of Petri nets. This approach has been extended by the author in cooperation with Jan Stückrath as described in Stückrath's thesis [265].

Before discussing the extension of Ehrig's approach to higher Petri nets on the basis of XML-based languages, the use of the double-pushout approach will be introduced by showing the cons of a single-pushout approach, especially in later implementation.

Theoretical Background

The use of the category theory approach for rule-based graph rewriting is mainly based on the single- (SPO) and double-pushout (DPO) approaches. The double-pushout approach was

introduced as part of a tutorial by Ehrig [71] and followed by the introduction of the SPO approach by Löwe [163]; the latter seems more elegant than the DPO approach, but problems can result from its less specific definition of transformation rules. The DPO approach combines categories of graphs with total graph morphisms, which offers a simple transformation of proofs and concepts to various types of graphs. The SPO approach, in contrast, simplifies the structure of transformation rules by only using one pushout, which results in a more complex category that also allows partial graph morphisms. This makes the SPO approach more expressive than the DPO approach, but also leads to more complex behavior of the transformation system and can thus result in a non-predictable outcome of the transformation. The two approaches are compared in greater detail in [73].

To reduce the complexity of the transformation to a minimum, in this dissertation, the DPO approach is used to reconfigure formal interaction logic. Thus, the DPO approach will be introduced based on a general graphical structure described in Definition 4.3 to stay close to the literature and to build a basis for its extension to the rewriting of reference nets. Some definitions from category theory will also be given; while they will not provide a complete discussion of category theory, but they offer a good basis for comprehending the following definitions.

Definition 4.3 (Graph) *A* directed graph *is a tuple* $G = (V, E, s, t)$ *with finite sets* V *and* E*, where* V *is a set of vertices or nodes and* E *is a set of edges.* $s : E \to V$ *and* $t : E \to V$ *are functions such that for any* $e \in E$*,* v_s *with* $s(e) = v_s$ *is the source node of* e *and* v_t *with* $t(e) = v_t$ *is the target node of* e*.* ◇

Based on this general definition of graphs, a first step toward defining rewriting systems can be made by formally defining the terms *production* and *matching*.

Definition 4.4 (Production and Matching) *A* production *is a mapping* $m : L \to G$*; a* matching *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.* ◇

The basic idea of rule-based graph rewriting is to define a production p that maps a graph L to a graph R, defining what has to be changed or how R is 'produced' from L. For instance, graph L has nodes that are not mapped to R. Therefore, these nodes will be deleted. Next, this production has to be matched to graph G, which is rewritten. To do this, different graphs can be applied to one production. Thus, a matching m determines which nodes on the left side L of the production p are matched to G and which parts are changed as determined by production p. When calculating p^* and m^*, H is also calculated by applying the production p to the matched parts of L in G defined by m. In general, only nodes are mapped to one another. Still, this is true only for definitions of graphs where edges are not defined as individual objects. In such cases, more specific mappings have to be defined to avoid inconsistencies. Such extensions will be examine below.

Based on Definition 4.4, the term *rule* for general graph rewriting can be defined as the combination of a production p, a matching m, and the graphs L and R as follows:

Definition 4.5 (Rule) *A* rule *is a tuple* $r = (m, p, L, R)$ *for transformation of a given graph* G*, where* $m : L \to G$ *is a* matching *and* $p : L \to R$ *is a* production*.* L *is*

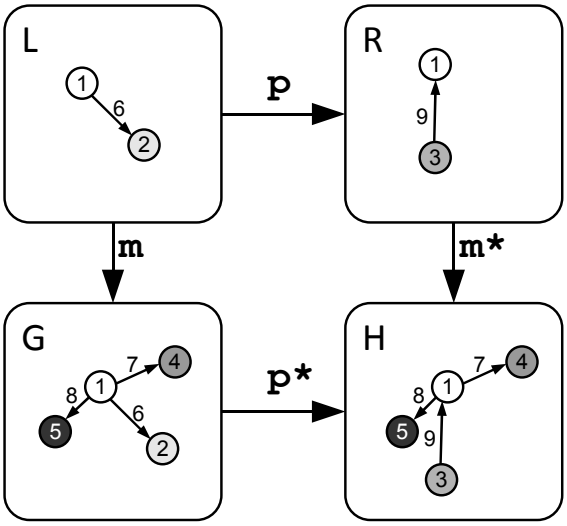


Figure 4.1.: Example of the application of a rule $r = (m, p, R, L)$ to a graph G resulting in a graph H

the left side of r, and R is the right side of r, where L and R are graphs. Application of a rule r is defined by replacement of the occurrence of L in G defined by m with R defined by p. ◇

Based on Definition 4.5, it is possible to define universal productions for various types of graphs, depending on an applicable choice of function m. Figure 4.1 shows an example of a rule $r = (m, p, L, R)$ transforming a graph G to graph H. Function m is implicitly defined by labeling the numbers of nodes and edges and identifying which node or edge in the rule's graphs (L and R) is matched to which node or edge in G and then in H, therefore, the numbers should not be interpreted as the weights of the nodes or edges. Thus, only nodes and edges with identical labels are matched to each other by the functions m, p, m^*, or p^*. For instance, p only matches node 1 in L to node 1 in R. No more nodes or edges in graph L or graph R are matched to one another. In Figure 4.1, associated nodes are also indicated by identical shade of gray. Thus, production p of the rule defines that edge 6 and node 2 have to be deleted from and node 3 and edge 9 added to graph G. Generally, the application of a rule r to a graph G means that

1. all nodes and edges in graph L that have an image in graph R defined by p will stay unmodified and will be part of H,

2. all nodes and edges in graph L that have no image in graph R will be deleted in G and will thus be missing in H, and

3. all nodes and edges in graph R that have no preimage in graph L will be added to G and thus will also be added to H.

Thus, all nodes and edges in graph G that do not have a preimage in L defined by the matching function m will stay unmodified. In this way, m defines which nodes and edges of G represent the occurrence of L in G. Because of this role of m as part of a rule r, it fulfills the requirements of being a (total) homomorphism. If m is a homomorphism, m is a function that

obtains the structure of the mapped graph. The fact that m is a homomorphism is especially important for the mapping of edges; an edge in G has to have the correct nodes as both source and target as is the case in L. Thus, given two graphs $L = (V, E, s, t)$ and $G = (V', E', s', t')$, for $m : L \to G$, it has to be true that

$$\forall e \in E' : s'(m(e)) = m(s(e)) \wedge t'(m(e)) = m(t(e)).$$

On the other hand, p does not necessarily have to be a total morphism. It is precisely this that differentiates the SPO from the DPO approach. But before discussing this difference in detail, the definition of this rule-based transformation of graphs must be embedded in category theory to offer a well-defined theoretical basis. Therefore, *pushouts* will be introduced formally as a well-known tool in category theory for talking about homomorphisms. Informally, a pushout is constructed of two category-theoretical arrows that are executed simultaneously (cf. [10]).

Definition 4.6 (Pushout) *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 $h \circ f^* = f'$.* ◇

Definition 4.6 is visualized in Figure 4.2. The first restriction $g^* \circ f = f^* \circ g$ applied to the arrows f and g and their counterparts f^* and g^* avoids the order of application of arrows to A, but does not change the result D. Thus, it does not matter how A is mapped to the pushout object D. Still, the choice of the pushout (D, f^*, g^*) for (f, g) is not unique. There could be various pushouts that fulfill the first restriction. Therefore, the second restriction implies that for one pushout only no arrow h exists, such that there is another object E that h is pointing to. That reduces the possible pushouts to exactly one, except in the case of isomorphism. Thus, defining (f, g) as a transformation rule, there is exactly one resulting pushout (D, f^*, g^*) that is the deliberate behavior of a rewriting rule.

In general, it is possible that two arrows (f, g) will not have a pushout. This is not the case for the category of graphs, which makes it possible to use pushouts to model transformation rules for graph rewriting. Given a rule (f, g) and a graph A, building the pushout generates the transformed graph as pushout object. That a pushout can be always calculated in the graph category results from a reduction of that category to the category of sets because transformation of nodes and edges can be handled separately. For more details, see [1, 10, 73].

To introduce the DPO approach, a further definition is necessary: that of the *pushout complement*.

Definition 4.7 (Pushout Complement) *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).* ◇

The *pushout complement* as defined in Definition 4.7 is part of the definition of rules for rewriting graphs in the DPO approach, as will be described in the next paragraph. It is essential for applying the transformation defined by a rule to a graph.

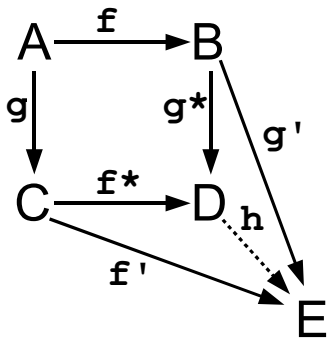


Figure 4.2.: Visual presentation of pushout

DPO Approach

The former definition of a rule $r = (m, p, L, R)$ as a tuple with two morphisms m and p and two graphs L and R can be transferred to the use of pushouts, as shown above. Therefore, m and p are considered arrows in category theory, where p need not be a total morphism. Building the pushout (D, m^*, p^*) to the pair (m, p), the pushout object D is the transformed graph. Thus, because p need not be a total homomorphism, r specifies a rule of the SPO. The problem with this approach is that the production is a partial homomorphism, which is likely to lead to an unclear result. Another option is the DPO approach, which replaces the single and partially defined production homomorphism with two total homomorphisms.

Definition 4.8 (DPO Rule) *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. ◇

The example seen in Figure 4.1 shows an SPO rule applied to a graph. The identical rule is shown as a DPO rule in Figure 4.3 and applied to the same graph. The difference is that deletion and addition operations are applied to graph G in two steps. In the first step, graph C is generated by determining the pushout complement (C, c, l^*) corresponding to (l, m). H results as the pushout object of the pushout (H, r^*, m^*) corresponding to (r, c). The pushout (H, r^*, m^*) is unique, resulting from its definition. Nevertheless, pushout complements do not have to be unique, but only to exist. For more information on this subject, see Ehrig et al. [73] and Heumüller et al. [110].

The problem of the nonexistence of pushout complements can be solved by introducing a further restriction to the creation of DPO rules called the *gluing condition*.

Definition 4.9 (Gluing Condition) *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*

$$\nexists e \in (E_G \setminus m(E_L)) \quad : \quad s_G(e) \in m(V_L \setminus l(V_I)) \vee t_G(e) \in m(V_L \setminus l(V_I)), \text{and} \tag{4.1}$$

$$\nexists x, y \in (V_L \cup E_L) \quad : \quad x \neq y \wedge m(x) = m(y) \wedge x \notin l(V_I \cup E_I). \tag{4.2}$$

◇

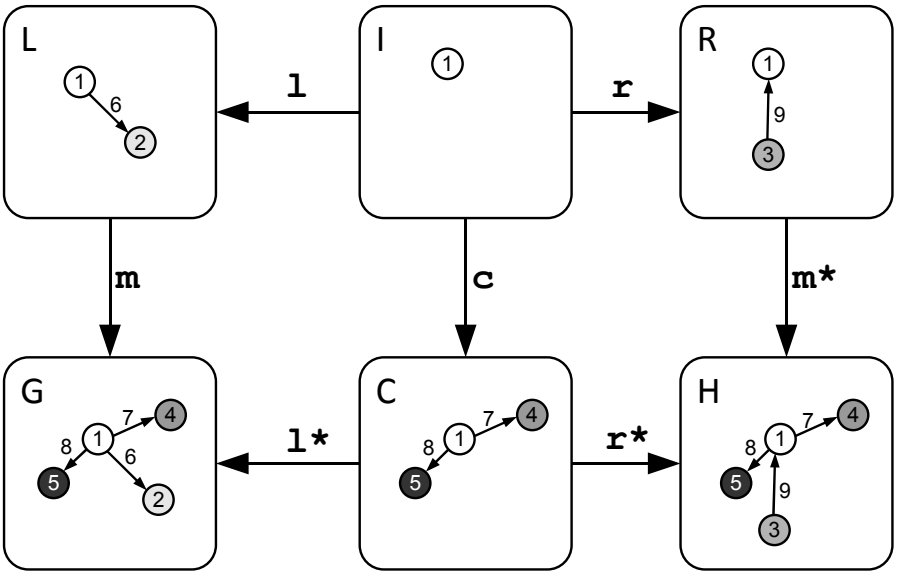


Figure 4.3.: Example of the application of a rule $s = (m, (l, r), L, I, R)$ to a graph G resulting in a graph H

In Definition 4.9, the gluing condition of two graph homomorphisms is defined using two assertions, the *dangling condition* in Equation 4.1 and the *identification condition* in Equation 4.2. The application of the gluing condition has already been indicated in the definition by choosing the identifiers l and m for the homomorphisms. When the gluing condition is applied to the left side of a DPO rule, the assertions have the meanings discussed below.

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.

Identification Condition The homomorphism m of a DPO rule that matches nodes and edges in L to nodes and edges in graph G 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.

Figure 4.4 shows these two basic problems concerning deletion using DPO-based graph rewriting (on the left side of a DPO rule). Pushout diagram (a) shows a violation of the identification condition. Here, nodes (0) and (1) in L are matched to only one node $(0, 1)$ in G. The identification condition is violated because $(0, 1)$ has two preimages in L. A brief idea of proof that no pushout complement exists in this case is that, when the rule (l, c) is applied, G is found not to be a pushout object. This is because there is no total morphism h that maps G to *all* other possible graphs G' that can be mapped by m' and l'. Still, this condition is essential if G is to be a pushout object of the pushout of (l, c), as defined in Definition 4.6. Otherwise, it is not a pushout.

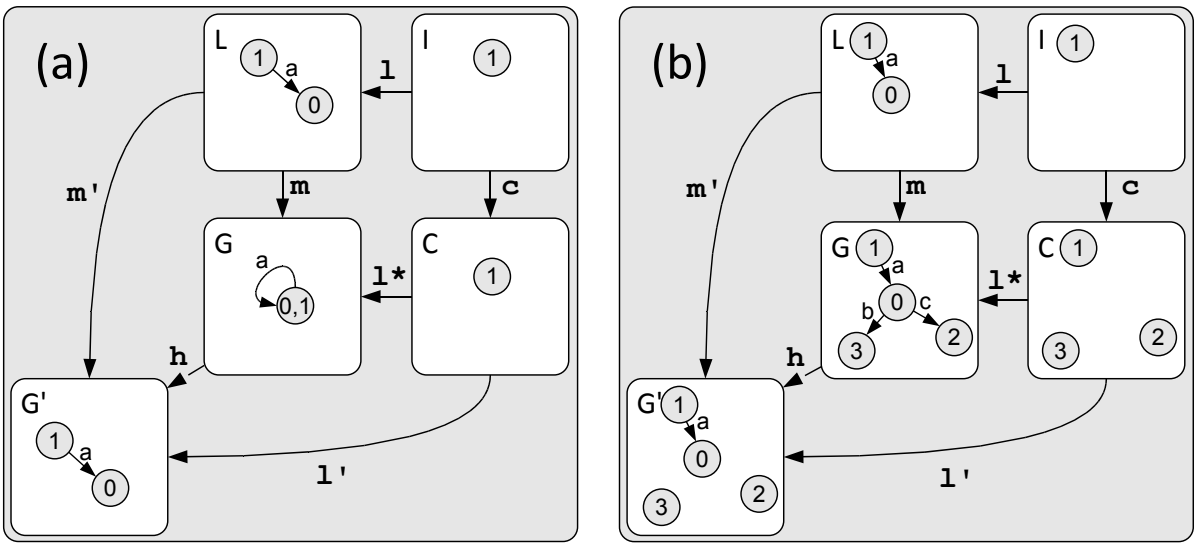


Figure 4.4.: Pushout diagram (a) shows an example of a violation of the identification condition; pushout diagram (b) shows an example of a violation of the dangling condition.

Another argument is that the correct example of a graph G' in Figure 4.4 (lower left) is possible, but no total morphism h' can exist that maps all the elements of G to G'. This problem can be traced back to the missing node (0) in C. Of course, there could be an example in which C contains a node (0), but with the given matching function m, G is still not a pushout because one instance of G' can be found where no total morphism h exists for mapping G to G'.

Figure 4.4 also includes pushout diagram (b), which shows a violation of the dangling condition. Here, node (0) will be deleted without defining what has to be done with the edges connecting nodes (2) and (3) to node (0). As seen above, a graph G' can be found such that G cannot be mapped to G' by the total morphism h; thus, in (b) no edges between (0), (2), and (3) can be matched to elements in G'. Here again, the missing nodes and edges in graph C can be identified as the reason this problem arises.

For the latter example, a correction can be applied to the rule as shown in Figure 4.5. Here, the surrounding nodes of node (0) are introduced to L. Now, a total morphism h can be found for an exemplary graph G'. Figure 4.5 shows a possible graph G', in which arrows m' and l' and a morphism h exist; however, G' is not 'minimal' because of node (4). That G is a pushout object of a valid pushout for (l, c) has to be proved showing that for all possible G' graphs mapped by the arrows m' and l', a morphism h exists. For further discussion and proofs, see Adamek et al. [1] and Corradini et al. [48].

Applying the gluing condition to the use of DPO rules solves the problem of the possible non-existence of a pushout complement. This is why the gluing condition has been included as an integral part of the DPO approach. Accordingly, DPO rules that do not fulfill the gluing condition are not applied to any graph.

Besides determining whether a pushout complement exists or not, there is still the problem that the pushout complement does not have to be unique. The pushout complement is unique if l and m are injective. If this is not the case, however, diverse correct pushout complements exist for a pair (l, m) of homomorphisms. Further investigation of this problem was conducted by Stückrath [265] and Heumüller et al. [110].

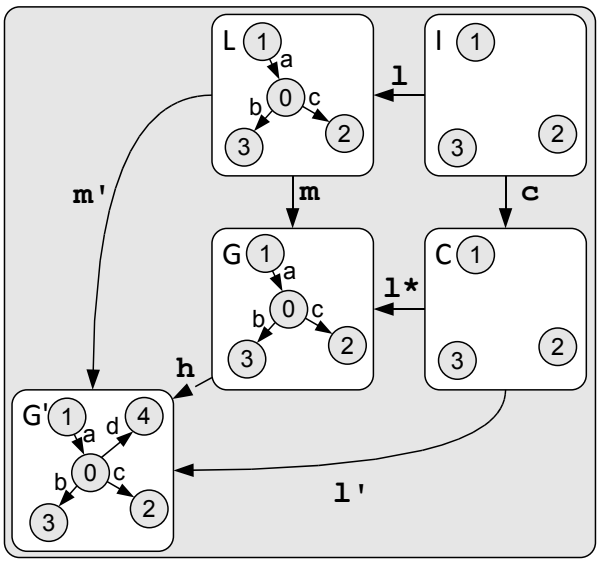


Figure 4.5.: Pushout diagram with fixed left side of a DPO rule.

For further work based on the above considerations, the DPO approach was used in the implementation of a graph rewriting system for the formal reconfiguration of interaction logic based on reference nets. For this purpose, DPO rules must meet the following two requirements:

1. It must fulfill the gluing condition to ensure the existence of a pushout complement of the left side of a DPO rule, and

2. in the DPO rule, l and m are injective to ensure the uniqueness of the pushout complement.

The following section describes how the general graph rewriting approach based on DPOs has been adapted for its use with reference nets.

Extension of DPO Approach to Reference Nets

Before the DPO approach can be extended to reference nets or, more generally, to colored Petri nets, it has to be transfered first to simple Petri nets, also called place transition nets (P/T nets). To that end, this section will first examine the work by Ehrig et al. [74, 75]. Then, it will explore work done in cooperation with Stückrath [265] extending the basic graph rewriting approach for P/T nets to colored and thence to reference nets.

Application of DPO to P/T Nets

Before looking at how Ehrig et al. extend graph rewriting on the basis of the DPO approach, it is necessary to define P/T nets.

Definition 4.10 (P/T net) *A* P/T net *is a graph defined as a tuple of four elements* $(P, T, pre, post)$. $p \in P$ *is called* place, $t \in T$ *is called* transition. *The functions* $pre : T \to P^{\oplus}$ *and* $post : T \to P^{\oplus}$ *define* edges *of a P/T net, where* $P^{\oplus}$ *is a set of all finite multi-sets generated with elements of* P. ◇

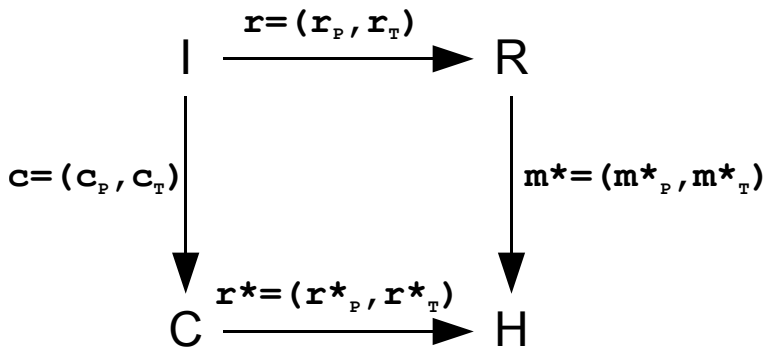


Figure 4.6.: Pushout diagram applying homomorphisms to P/T nets

As shown above, applying a DPO rule to a graph or net G means determining the pushout (object) of the right side of the rule and the pushout complement (object) of the left side of the rule. Before introducing the determination of pushouts on homomorphisms for P/T nets, homomorphisms on P/T nets need to be defined. Here, the main problem to be solved is the extension of homomorphisms to associate two different types of nodes in two graphs with one another: places to places and transitions to transitions.

Definition 4.11 (Homomorphisms on P/T nets) *A homomorphism $r : I \to R$ on two P/T nets $I = (P_I, T_I, pre_I, post_I)$ and $R = (P_R, T_R, pre_R, post_R)$ is a pair (r_P, r_T) of two homomorphisms with $r_P : P_I \to P_R$ and $r_T : T_I \to T_R$ where for all $t \in T_I$*

$$\begin{aligned} pre_R(r_T(t)) &= r_{P^\oplus}(pre_I(t)), \text{ and} \\ post_R(r_T(t)) &= r_{P^\oplus}(post_I(t)) \end{aligned}$$

is true. $r_{P^\oplus} : P_I^\oplus \to P_R^\oplus$ is a function that maps a multiset of elements from P_I to a multiset of elements of P_R such that

$$\bigcup_{p_I \in P_I^\oplus} \{r_P(p_I)\} = P_R^\oplus. \qquad \diamond$$

The requirements to r_P and r_T defined in the above definition guarantees that the characteristic structure for homomorphisms is maintained. This kind of 'split definition' of a homomorphism $r = (r_P, r_T)$, mapping two P/T nets to one another, offers a handy way to calculate pushouts on P/T nets based on the above-introduced approach for general graphs providing only one type of node. Figure 4.6 shows a pushout diagram of the right side of a DPO rule using split homomorphisms on P/T nets. H should be calculated as a pushout object of the pushout of the pair of homomorphisms (r, c) and the P/T nets I, R, and C, as is the case in the graph rewriting system using the DPO approach.

To determine the pushout shown in Figure 4.6, it is enough to determine the pushout in the category of sets of (r_P, c_P) for places and the pushout in the category of sets of (r_T, c_T) with pushout object P_H and T_H. As described for pushouts in general and in their application in graph rewriting, P_H and T_H are determined by gluing P_R and P_C through P_I together, and T_R and T_C through T_H respectively, such as

$$\begin{aligned} P_H &:= P_C +_{P_I} P_R, \text{ and} \\ T_H &:= T_C +_{T_I} T_R. \end{aligned}$$

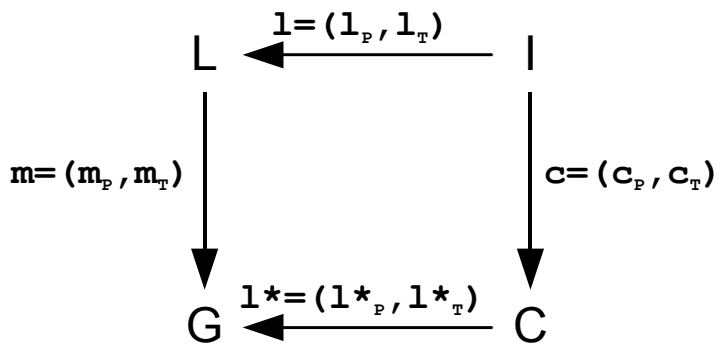


Figure 4.7.: Pushout diagram showing the left side of a DPO rule.

Next, pre_H and $post_H$ of net H have to be specified. This is slightly different from the above definition of pushouts in graphs in general. Here, edges are defined implicitly and not as individual objects. pre_H and $post_H$ can by

$$pre_H(t) = \begin{cases} m^*_{P\oplus}(pre_R(t_j)), \text{ if } m^*_T(t_j) = t \text{ and } t_j \in T_R \\ r^*_{P\oplus}(pre_C(t_k)), \text{ if } r^*_T(t_k) = t \text{ and } t_k \in T_C \end{cases}, \text{ and}$$

$$post_H(t) = \begin{cases} m^*_{P\oplus}(post_R(t_j)), \text{ if } m^*_T(t_j) = t \text{ and } t_j \in T_R \\ r^*_{P\oplus}(post_C(t_k)), \text{ if } r^*_T(t_k) = t \text{ and } t_k \in T_C \end{cases}.$$

If this implicit definition of edges in a P/T net is taken into consideration, it becomes clear that the explicit definition of adding or deleting edges cannot be specified in transformation rules using the approach described above. Instead, adding or deleting edges is implicitly defined through its connected transition. This means that if an edge should be deleted, but the associated transition should not, the transition must first be deleted and then added back into the net without the unwanted edge. The process is analogous for adding edges. This restriction makes the application of rules safer in the sense that adding an edge to or deleting an edge from a transition means changing the meaning of the transition. Still, in some cases this change in the semantics of a net can be helpful or even desirable. Adding and deleting edges will be part of the discussion of extending the DPO approach to colored nets in the next section.

The next step will be to describe how the pushout complement is determined in order to calculate the parts of the net to be deleted. Central to determining the pushout complement is the gluing condition, which guarantees the existence of the pushout complement if it is applied to the rules of the formalism (here, DPO rules on P/T nets). In this case, the left side of a DPO rule is of interest because it is on this side that the pushout complement has to be calculated. The left side of a DPO diagram can be seen in Figure 4.7.

The transformation of the gluing condition to the application of the DPO approach to P/T nets can be conducted by

$$GP = l(P_I \cup T_I), \tag{4.3}$$

$$IP = \{p \in P_L | \exists p' \in P_L : (p \neq p' \wedge m_P(p) = m_P(p'))\} \tag{4.4}$$
$$\cup \{t \in T_L | \exists t' \in T_L : (t \neq t' \wedge m_T(t) = m_T(t'))\}, \text{ and}$$

$$DP = \{p \in P_L | \exists t \in T_G : \nexists t' \in T_L : m_T(t') = t \wedge (m_P(p) \in pre_G(t) \vee m_P(p) \in post_G(t))\}. \tag{4.5}$$

These are based on the homomorphism and nets shown in Figure 4.7. The set GP defined in Equation 4.3 is a set of all places and transitions that are mapped from net I to net L by the

morphism l. Furthermore, GP is a set of places and transitions that are part of net L and are images of the nodes in I concerning l. Thus, these nodes should not be deleted from G. As it is known, the *identification condition* is fulfilled if no node exists that is deleted $((P_L \cup T_L) \setminus GP)$ and simultaneously mapped to more than one node in G. Therefore, all nodes of L that are mapped to more than one node in G are collected in set IP. Thus, a DPO rule fulfills the identification condition if for the sets GP and IP, $IP \subseteq GP$ is true.

A rule fulfilling the *dangling condition* has to require that no place that is connected to a transition in G can be deleted if it is not an image of a node in L defined by m. Therefore, DP contains all places that are mapped from L to G and are simultaneously connected to a transition. Thus, the dangling condition is fulfilled if $DP \subseteq GP$ is true for a given DPO rule. The dangling condition for transitions does not have to be formulated explicitly because of the implicit definition of edges. If a transition is removed from the net, the connected edges are also removed.

Under the assumption that the gluing condition is fulfilled by the left side of a given DPO rule, the pushout complement object C can be generated by

$$\begin{aligned} P_C &= (P_G \setminus m_P(P_L)) \cup m_P(l_P(P_I)), \text{ and} \\ T_C &= (T_G \setminus m_T(T_L)) \cup m_T(l_T(T_I)), \end{aligned}$$

where $m_P(P_L)$ represents the image set of m_P similar to $m_T(T_L)$ representing the image set of m_T. The morphisms pre_C and $post_C$ result from the reduction of their domain to T_C. It is also true that $pre_C(t_C) = pre_G(l_T^*(t_C))$ and $post_C(t_C) = post_G(l_T^*(t_C))$.

Extension to Colored Petri Nets

Extending the DPO approach for P/T nets to colored Petri nets mainly involves extending the rewriting by adding inscriptions to the rewriting rules. Furthermore, associating inscriptions with edges requires that edges be defined as independent objects, as transitions and places are. Extending the DPO approach in this context is similar to the process shown in the previous section, where basic graphs offering only one type of node were extended in P/T nets to include two types of nodes.

Before continuing, it is necessary to define colored Petri nets. This unspecific definition of higher Petri nets is complex enough to investigate the rewriting of Petri nets involving inscription languages and simple enough to avoid complex formalization that is only applicable to one specific type of Petri net and its associated inscription language. Still, all higher formalisms regarding Petri nets are extended by a special type of inscription language as is the case for reference nets. It has also to be noted that the following formalization does not take the semantics of the inscription language into account. This would also lead to a specification of rewriting, which is not the goal of this dissertation.

Definition 4.12 (Colored Petri Net) *A* colored Petri net *is a 6-tuple*

$$(P, T, E, I, e, i)$$

with P a set of places*, T a set of* transitions*, and E a set of* edges *where P, T, and E are pairwise disjoint. I is a set of all possible* inscriptions *based on a certain inscription language including the empty inscription ϵ. $e : E \to (P \times T) \cup (T \times P)$ is a total function defining the source and target node of an edge. $i : (P \cup T \cup E) \to I$ is a total function mapping an inscription to a place, transition, or edge.* ◇

As discussed above, a colored Petri net can be distinguished from basic P/T nets in two important ways: (a) edges are defined as individual elements and are thereby explicitly defined, and (b) inscriptions can be associated with places, transitions, and edges, thus extending the formalisms and making them more flexible, but not necessarily more powerful. The latter assertion depends primarily on the type of inscription language and its structure, which will not be dealt with here. For further discussion, please see the books [125, 126] by Jensen, which describe a specific type of colored Petri nets and suggest further reading. One example of an inscription language is that of reference nets based on the Java type system as introduced by Kummer [149] and explained in Chapter 3 above.

As for P/T nets, for colored nets the definition of a graph homomorphism $f : A \to B$ mapping the nets A to B can be extended. The problem here is not to maintain the net's structure by defining how transitions are mapped and the characteristics of the associated edges' defining functions (*pre* and *post*). Here, the problem is the mapping of nodes and edges to each other that are additionally inscribed with more or less complex expressions. Inscribed nodes and edges cannot be handled in the same way as in the above definitions because in colored nets, each node is individualized through its inscription. Therefore, mapping two graphs is correct if and only if the image node's inscription is identical to that of the preimage node. This assumption makes it possible to neglect further considerations concerning homomorphisms in the context of inscriptions. Thus, the factor of inscriptions is kept out of the further extension of DPO to colored nets.

Definition 4.13 (Total Homomorphisms on Colored Nets) *A total homomorphism $r : A \to B$ on two colored nets $A = (P_A, T_A, E_A, I, e_A, i_A)$ and $B = (P_B, T_B, E_B, I, e_B, i_B)$ is a triple (r_P, r_T, r_E) of three total homomorphisms with $r_P : P_A \to P_B$, $r_T : T_A \to T_B$, and $r_E : E_A \to E_B$. For r, the following restriction has to be fulfilled, where $fst_X : E_X \to (P_X \cup T_X)$ maps an edge of net X to its source element and $lst_X : E_X \to (P_X \cup T_X)$ maps it to its target element, where any edge is directed and points from its source to its target, such as*

$$\forall x \in (P \cup T \cup E) \quad : \quad i_A(x) = i_B(r(x)), \tag{4.6}$$

$$\forall e \in E_A \quad : \quad r(fst_A(e)) = fst_B(r_E(e)), \text{ and} \tag{4.7}$$

$$\forall e \in E_A \quad : \quad r(lst_A(e)) = lst_B(r_E(e)). \tag{4.8}$$

◇

The Restrictions 4.7 and 4.8 in Definition 4.13 guarantee the structure maintenance of function r_E, making r a homomorphism. Restriction 4.7 states that for all edges e in the preimage graph A, the image of its source nodes has to be identical to the source node of the image of e in graph B. The same is true for Restriction 4.8, which says that the image of the target element of an edge e in graph A has to be identical with the image of the target element of the image of edge e in B. Figure 4.8 provides two examples of the mapping of graph A to B. The upper one shows a violation of Restriction 4.7, where all nodes and edges of graph A are mapped to nodes and edges in B, but the image of the source node of edge e (labeled $1'$) is not the source node (labeled 1) of the image e' of edge e in B. The lower example shows a violation of Restriction 4.8, where the image of the target node (labeled 1) in graph A of edge e (labeled $1'$) is no longer the target node of the image e' of edge e. Equation 4.6 specifies how to handle inscriptions. Here, the inscription of an element in the preimage graph has to

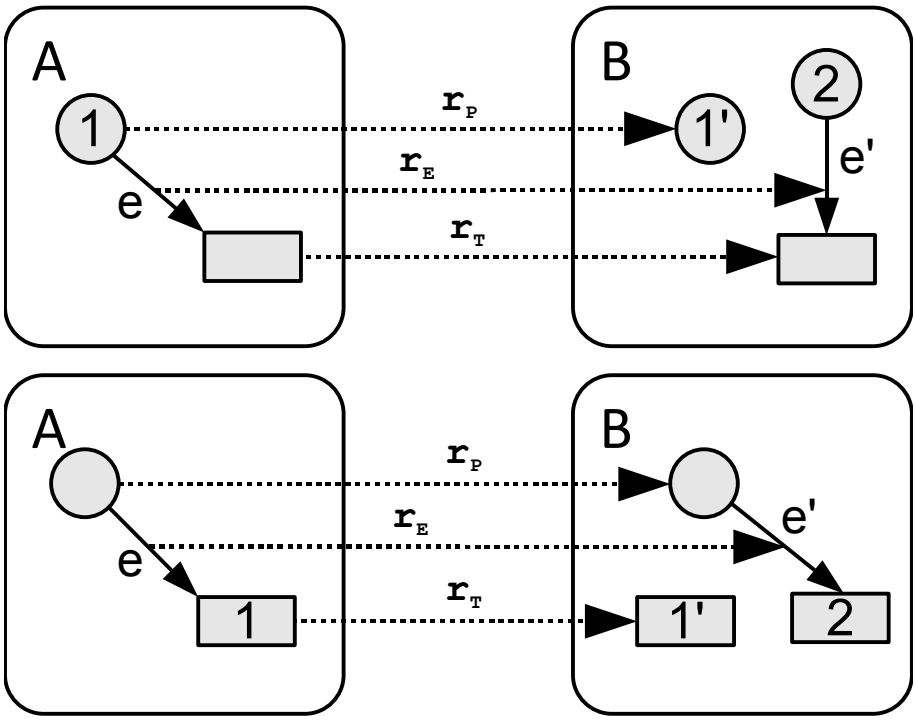


Figure 4.8.: Violation of Restriction 4.7 (above) and of Restriction 4.8 (below)

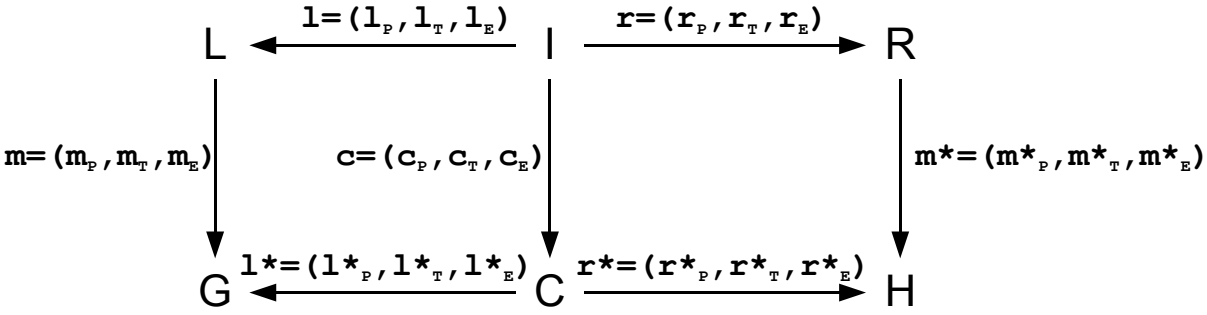


Figure 4.9.: Pushout diagram showing a DPO rule for colored Petri nets

be identical with the inscription of its image. Later on, it will be shown how this point can be applied to simplify the formulation of transformation rules offering the transformation of inscriptions. In the following, only r is assumed to include r_P, r_T, and r_E as long it is clear which partial morphism is relevant in a given context.

Figure 4.9 shows a DPO rule for colored nets using a total homomorphism as specified for the DPO approach. The pushout object H on the right side of the rule is determined by gluing together P_R and P_C through P_I, T_R and T_C through T_I, and E_R and E_C through E_I, such as

$$\begin{aligned} P_H &:= P_C +_{P_I} P_R, \\ T_H &:= T_C +_{T_I} T_R, \text{ and} \\ E_H &:= E_C +_{E_I} E_R. \end{aligned}$$

Based on the definition of the homomorphisms for colored Petri nets, it is not necessary to specify further restrictions. The structure of e_H and i_H directly results from Equations 4.7, 4.8

for e_H, and 4.6 for i_H.

To determine the pushout complement for the left side of the rule shown in Figure 4.9, first the gluing condition has to be adapted to colored Petri nets. For that purpose, the sets GP, IP and DP introduced above can be extended for colored Petri, such as

$$GP = l(P_I \cup T_I \cup E_I), \tag{4.9}$$

$$IP = \{x \in (P_L \cup T_L \cup E_L) \,|\, \exists x' \in (P_L \cup T_L \cup E_L) : (x \neq x' \wedge m(x) = m(x'))\}, \text{ and} \tag{4.10}$$

$$DP = \{x \in (P_L \cup T_L) \,|\, \exists e \in E_G : (m(x) = \text{fst}_G(e)) \vee m(x) = lst_G(e))) \wedge \nexists e' \in E_L : m(e') = e\}. \tag{4.11}$$

The set GP contains all places, transitions and edges that should not be deleted from G. The representation of edges as individual objects makes it necessary to identify the fusion of edges as was the case for places and transitions in the extension to P/T nets. Thus, IP is extended with all edges in L that are matched by m to a image edge in G. Now, a DPO rule for colored Petri nets fulfills the *identification condition* if $IP \subseteq GP$ is true. The set DP need not to be modified and the *dangling condition* is fulfilled if $DP \subseteq GP$ is true for a given DPO rule.

If the gluing condition is fulfilled by a given DPO rule, the pushout object C of the pushout complement of (l, m) can be calculated by

$$\begin{aligned} P_C &= (P_G \setminus m_P(P_L)) \cup m_P(l_P(P_I)), \\ T_C &= (T_G \setminus m_T(T_L)) \cup m_T(l_T(T_I)), \text{ and} \\ E_C &= (E_G \setminus m_E(E_L)) \cup m_E(l_E(E_I)). \end{aligned}$$

e_C and i_C result from a reduction of their domains.

The Problem of Inscription Rewriting

The ability to inscribe nodes and edges is one of the most important aspects of colored Petri nets. The sections above cover the adaption of the DPO approach to colored Petri nets, but do not address the topic of rewriting inscriptions. However, DPO rules also have to cover inscriptions, in that L, I, and R also have inscribed nodes and edges. This is nothing new as the basic theory expects that the rule graphs will have the same syntax (and semantics) as the graph to be rewritten by the rule. Otherwise, the rewriting approach would not be able to change the whole range of semantics in the given graph-based formalism. For instance, rule graphs without inscriptions would only be able to add nodes and edges to the graph with empty inscriptions, which would also require the colored Petri net formalism to allow for nodes and edges with empty inscriptions (as most formalisms of this kind do). A further problem would be matching graph L to G by m. If m did not take the inscription into account, the match probably could not been made correctly when applying the rule because the inscriptions in graph G are essential elements defining its semantics. Thus, a match that does not consider inscriptions is not able to cover the entire possible range of a rewriting formalism.

Considering the basic theory of pushouts as used in the DPO approach, the problem of inscriptions in graph rewriting becomes even more critical. As shown in Figure 4.2 and defined in Definition 4.6, the pushout object D has to be unique (except in the case of an isomorphism) such that for all graphs E, there is an arrow h. In the case of graphs with inscriptions that are not considered by the homomorphisms in a pushout diagram, it is possible to construct a graph E which is isomorphic to graph D and thus 'equal' in the sense of the category used, but which provides different inscriptions for nodes and edges. Going back to the original definition

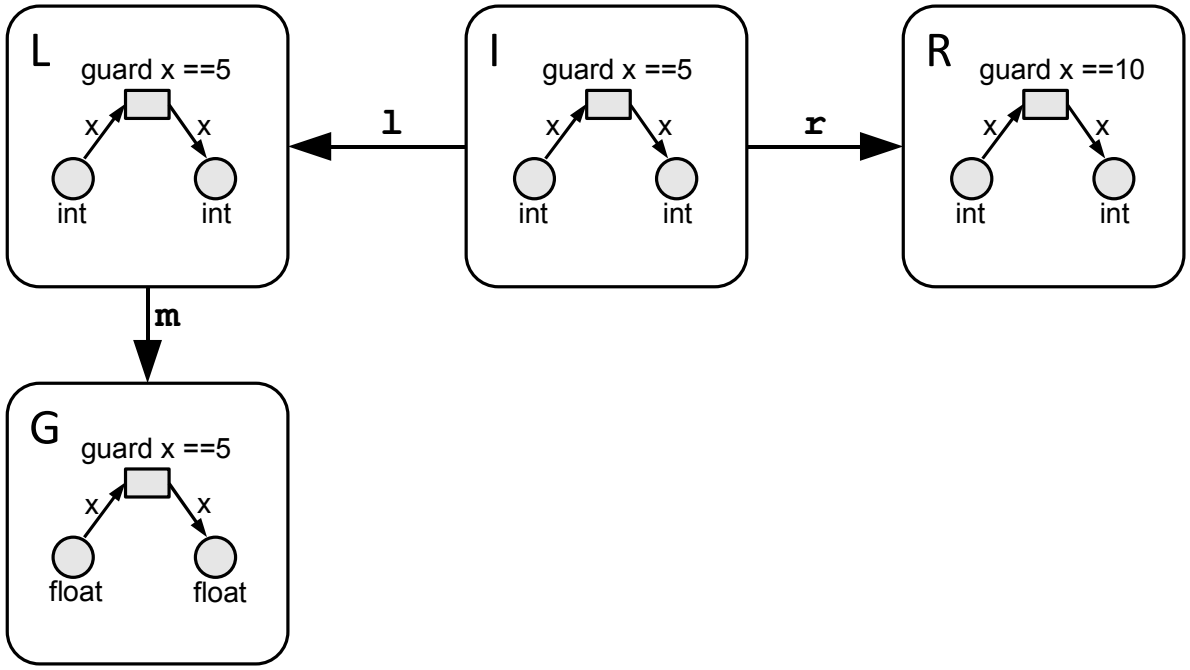


Figure 4.10.: Example of the application of an order relation to the matching function m of a given DPO rule with direct rewriting of the inscription

of the syntax and semantics of the graph-based formalism being considered, D and E show different syntax and semantics and therefore should be identified as different types of graphs, but the pushout does not do so. This would lead to the problem that the diagram supposed to be a pushout is not a pushout at all because D is not unique when the syntax and semantics definition of the colored Petri net formalism is taken into account. These circumstances lead to the decision that inscriptions have to be considered in the context of DPO rules so as not to violate the underlying theoretical concepts of pushouts and, in this way, risk an inconsistent solution.

The straightforward solution for this problem is to extend the mapping functions m, l, and r so that they also take into account the inscriptions of the places, transitions, and edges of all the nets, such that, for instance, a place only matches another place if their inscriptions are equal, as it has been defined in Definition 4.13. This kind of restriction influences the possible matches to the graph G, as well as the possible rewrites defined by l and r with graphs L, I, and R. In fact, if only unchanged inscriptions are allowed, it is not possible to rewrite inscriptions. Still, inscriptions are important for the semantics of a given graph and should therefore be rewritable.

As proposed by Stückrath [265], matching inscriptions, especially in case of the matching function m, could be defined as a (partial) order relation $\sqsubseteq$, such that

$$\forall x \in (P_L \cup T_L \cup A_L) : incr_L(x) \sqsubseteq incr_G(m(x)). \tag{4.12}$$

This approach is an initial step toward rewriting inscriptions using the DPO approach. In some cases, this definition of an order relation has to be further investigated due to the definition of the semantics of the graph formalism to be rewritten. In the example shown in Figure 4.10, the matching is only correct if all the operations on integers also work with float values. Obviously, this is not the case. For instance, the function ggt, which calculates the greatest common divisor,

would not work on a float value because it is specifically defined as an integer-based function. Regarding this aspect of defining semantics, the specification of an order relation like the one in Equation 4.12 becomes complicated. Furthermore, there is no theoretical foundation for this approach even though it makes sense in the context of the application in formal reconfiguration of user interfaces and in general to make graph rewriting more flexible for use with inscribed graph formalisms.

However, it is not the main goal of this work to describe theoretical foundations for these kinds of problems, but that of future work. Still, a solution is needed to make the rewriting mechanism more flexible in its use for formal user interface reconfiguration. In this context, the next step is to find a data structure that offers a handy formulation and serialization of DPO rules to be applied to reference nets as well as a suitable handling of the 'rewriting inscription problem' using a particular partial order relation. Therefore, an XML-based approach will be introduced and discussed based on Stückrath's work [265].

Graph Rewriting through XML-based Petri Net Serialization

In the formal approach to graph rewriting and its application to colored Petri nets, it is necessary to examine its programmatic implementation. The implementation of a framework for the interactive and formal modeling of user interfaces paired with their simulation and reconfiguration is discussed in Chapter 5. Still, an XML-based representation of Petri nets in the context of graph rewriting and thus in the formal reconfiguration of interaction logic is closely related to the described DPO approach and its formal background. Therefore, the XML-based implementation of the DPO approach will be explored at this point and not as part of the implementation description in Chapter 5.

Various types of XML-based serialization of graphical data structures can be found on the Web. For instance, one well-known example is GraphXML, introduced by Herman and Marshall [109] and later renamed GraphML by Brandes et al. [33]. For Petri nets, with their diverse types of extensions in various research disciplines, an exchange format based on XML was first published by Jüngel et al. [129], extended by Weber and Kindler [279], and finally standardized as part of the ISO standard ISO/IEC 15909, described by Hillah et al. [112]. This standardization and its wide distribution in diverse implementations for modeling and simulating different types of Petri nets make PNML the best format for serializing interaction logic as reference nets and the most useful basis for describing DPO rules for rewriting reference nets.

For the implementation of a software system for graph rewriting of colored Petri nets, Stückrath [265] extended the basic structure of PNML as a RelaxNG schema file[1] (cf. [272]) (ISO/IEC 19757-2[2]) to an exchange format for colored nets like those introduced above. For this extension, RelaxNG offers concepts like inheritance and overwriting. Using the extension, reference nets can also be serialized to a PNML-based format. It is not the goal of the following descriptions to give a complete description to RelaxNG; instead, the aim is to sketch the implemented style file for describing colored Petri nets based on the version of PNML newly introduced by Stückrath. Nevertheless, the style files are part of the implementation and can be found in Appendix A.

In addition to defining colored nets as a PNML file, it is important to define an XML-based format for specifying DPO rules to be applied to a given reference net or graph in PNML

[1] http://www.pnml.org/version-2009/grammar/pnmlcoremodel.rng
[2] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=52348

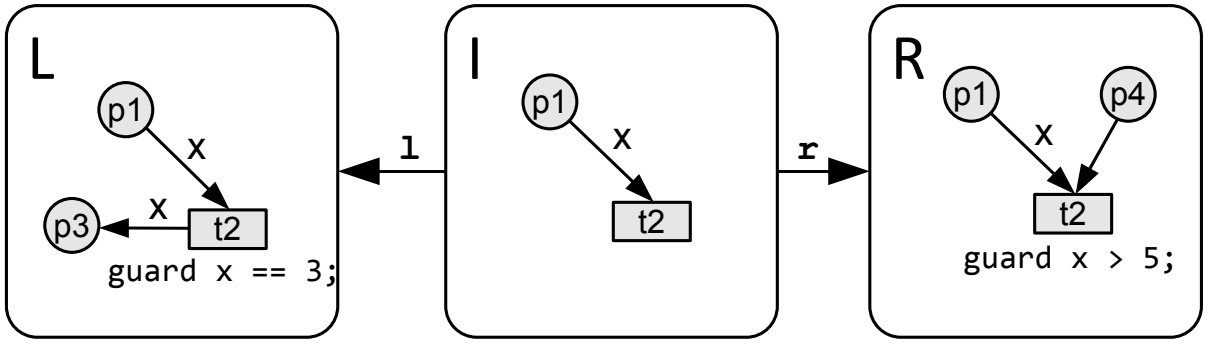


Figure 4.11.: Example of a production of a DPO rule for rewriting colored Petri nets

format. In general, a graph rewriting rule was defined as a combination of production(s) and a mapping function. In the context of this dissertation, the DPO approach was implemented because it can be easily verified by checking rules for fulfilling the gluing condition; thus, the correctness and applicability of a rule to a net can be determined such that an estimated result can be calculated and a result actually exists.

A DPO rule s was defined as a tuple $s = (m, (l, r), L, I, R)$ of a matching function m, a production (l, r) given as a pair of two total homomorphisms and three nets or graphs L, I, and R where L is called *left side*, R is called *right side*, and I is called *interface net* or *interface graph*. Mapping this formal definition of a DPO rule to an XML-based representation in the context of colored Petri nets results in the structure seen in Listing 4.12. Here, the XML-based format will be used to represent only the production and its necessary graphs L, I, and R, but not the matching function m. The reason for this is that once a DPO production has been generated, it can be applied to different nets without the necessity to change the content of the XML file representing the rule. The matching function is provided as separate information to the transformation engine that interprets the rule and applies it to a graph. The example in Listing 4.12 shows a production of a DPO rule as an XML file.

Related to Listing 4.12, a net is described using PNML by defining the root node `net` followed by its children, which define transitions, places, and edges represented as `transition`, `place`, and `arc` nodes. Any child of node `net` has an *id* and optional children defining inscriptions represented as `inscription` nodes. The production (l, r) is a `mapping` node that contains `mapElement` nodes that represent the mapping of elements in net L (`deleteNet`), I (`interface`), and R (`insertNet`). This way of modeling l and r in XML is possible because both morphisms are total and thereby have the same domain. Finally, it should be noted that only the nodes of type `net` and their child nodes are part of PNML. All other nodes are added to represent a DPO rule as an XML file in a consistent way.

As can easily be seen in Figure 4.11, one basic restriction on homomorphisms for colored nets has been violated: The inscriptions of the transition labeled `t2` differ between nets L, I, and R. This clearly relates to the problem of rewriting inscriptions as was forbidden by the definition of homomorphism given above for colored Petri nets. Still, it should be easily possible to rewrite inscriptions to simplify the use and implementation of this approach in a modeling, simulation, and reconfiguration framework based on it. Therefore, the following definition offers an extension using the XML-based formalization of Petri nets, which makes it possible to rewrite the inscriptions in Figure 4.11, defining a partial order relation as described above on the basis on the XML-based tree structure.

```
<rule>
  <deleteNet>
    <net id="net1" type="refNet">
      <transition id="t2">
        <inscription><text>guard x == 3</text></inscription>
      </transition>
      <place id="p1" />
      <place id="p3" />
      <arc id="e1" source="p1" target="t2">
        <inscription><text>x</text></inscription>
      </arc>
      <arc id="e2" source="t2" target="p3">
        <inscription><text>x</text></inscription>
      </arc>
    </net>
  </deleteNet>
  <interface>
    <net id="net2" type="refNet">
      <transition id="t2">
        <inscription><text>guard x == 3</text></inscription>
      </transition>
      <place id="p1" />
      <arc id="e1" source="p1" target="t2">
        <inscription><text>x</text></inscription>
      </arc>
    </net>
  </interface>
  <insertNet>
    <net id="net3" type="refNet">
      <transition id="t2">
        <inscription><text>guard x == 3</text></inscription>
      </transition>
      <place id="p1">
        <inscription><text>int</text></inscription>
      </place>
      <place id="p4" />
      <arc id="e1" source="p1" target="t2">
        <inscription><text>x</text></inscription>
      </arc>
      <arc id="e3" source="p4" target="t2" />
    </net>
  </insertNet>
  <mapping>
    <mapElement interfaceID="p1" deleteID="p1" insertID="p1" />
    <mapElement interfaceID="t2" deleteID="t2" insertID="t2" />
    <mapElement interfaceID="e1" deleteID="e1" insertID="e1" />
  </mapping>
</rule>
```

Figure 4.12.: Example of a DPO rule for rewriting colored Petri nets given in PNML format.

Definition 4.14 (XML Node Part of Relation) *An XML node A is* part of *another XML node B if the following two conditions are fulfilled:*

1. *B contains any attribute that A contains with the same values.*
2. *A has no child nodes* **OR**

   for any child A' of A there is a child B' of B such that A' and B' are identical or A' is part of B'.

Two nodes are identical if node A is part of a node B and B is part of A. ◇

This extension of mapping nets onto each other softens the restrictions made to homomorphisms on colored Petri nets (cf. Definition 4.13). Allowing the rewriting of inscriptions based on the *part of* relation for XML nodes as defined in Definition 4.14 makes it possible to change the semantics of transitions as occurs when the rewriting of edges alters the firing of transitions. As will be shown below, changing existing parts of a net to restrict such factors as token play is part of reconfiguration. Therefore, XML-based net rewriting using the *part of* relation is a helpful extension for later implementation. The main formal restrictions to the pushouts and pushout complements, like the gluing condition, are left untouched by this adaption of the formalisms.

Coming back to the example shown in Figure 4.11, using the *part of* relation, it is now possible to map the graphs to one another. This is because the transition's inscription with id `t2` in net I is part of the inscription attached to the transition `t2` in graph L. This is also true for its transformation to graph R. By extending the *part of* relation to XML nodes representing elements of the net, for instance, transitions, it is possible to validate the correctness of the mapping node as a rule. It is possible to check, for instance, whether, when one transition is mapped to another transition, the latter's inscriptions are *part of* the first or not, and so forth.

Taxonomy of Reconfigurations

The above formal introduction to the rewriting of colored Petri nets and their representation in XML-based formats offers a toolbox for modeling the reconfiguration of formal interaction logic as a reference net. Next, an overview of the possible reconfigurations of interaction logic will be introduced and described. To that end, a classification of reconfiguration will be proposed and provide a basis for further investigation into the reconfiguration of interaction logic. In Chapter 5, an algorithm for generating DPO rules for rewriting a formal interaction logic will be introduced that offers all the necessary information and tools for generating specific rewriting based on the informal introduction of reconfiguration operations below.

In the following descriptions, reconfiguration operations for application to interaction processes that process data inputted by the user are introduced independently from those for reconfiguring interaction processes that process data emitted from the system for presentation to the user. This differentiated presentation of reconfiguration operations offers a more understandable introduction to the problem because in many cases of reconfiguration operations, input and output interaction processes differ due to the type of data they process (controlling something or presenting the system state) and how they are triggered. Furthermore, it is easier to introduce these different classes of reconfiguration operations separately. Still, simple reconfiguration operations, like adding an interaction process to or deleting one from the interaction logic are identical. Also, some classes of reconfiguration operations are somewhat similar, like

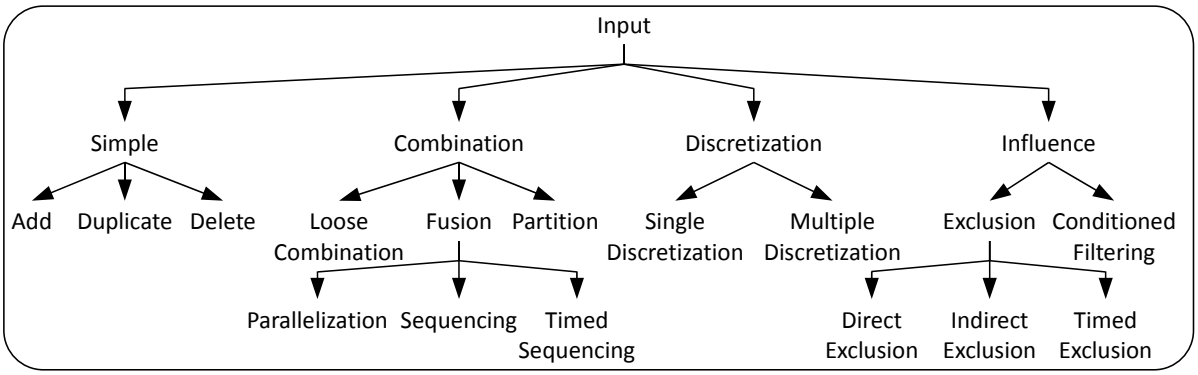


Figure 4.13.: Taxonomy of possible reconfigurations of formal interaction logic—Reconfiguration of input data processing interaction processes

combining specific interaction processes. Nonetheless, it is important to recognize that combining interaction processes for inputting data and combining those for outputting data differ in how they are combined.

In addition to reconfiguring the interaction logic, it is often necessary to redesign the physical representation. For instance, deleting an interaction process often involves deleting the associated interaction elements from the physical representation. This is also true for the deletion of interaction elements that are associated with interaction processes. To prevent inconsistency between the interaction logic and the physical representation, rules are specified in the detailed descriptions below. In general, the following directives can be introduced for any reconfiguration involving redesign.

Changing interaction process/interaction element For every change in interaction logic, the physical representation must be checked for compatibility: The changed interaction logic must still generate correct data to be presented by the interaction elements and must still be compatible with the data generated by the associated interaction elements.

Deletion interaction process/interaction element If an interaction process is deleted, it is necessary to ascertain whether the related interaction element is still needed or not. Similarly, if an interaction element is deleted, it is necessary to ascertain whether the associated interaction process is still needed or not, for instance, if it is connected to other interaction processes.

Adding interaction process/interaction element Here, it is necessary to ensure that an added interaction process is triggered by a interaction element and/or that it triggers an interaction element. Conversely, when an interaction element is added to the physical representation, it is necessary to ensure that it is connected to a interaction process. Also, the compatibility of data passed to and from interaction elements has to be verified as described above (due to changing interaction process/interaction element).

Thus, a consistent reconfiguration of a user interface guarantees that every interaction element is associated with a minimum of one interaction process and that no data type incompatibility arises. Conversely, consistent reconfiguration also guarantees that an interaction process is still connected to a minimum of one interaction element and/or interacts with other interaction processes.

Figures 4.13 and 4.15 show a possible selection of reconfigurations organized in a hierarchical taxonomy building classes of reconfiguration operations. This selection of operations is not complete, but offers a basic catalog of operation classes for reconfiguring interaction logic. A closer description of all classes follows. First, reconfiguration operations for adapting interaction processes for input data will be presented and discussed, followed by reconfiguration operations for adapting processes for output data. In this context, the necessary redesign of the physical representation will be briefly described. This description will address only the addition and deletion of interaction elements, always assuming that no data compatibility problems arise from applying the reconfiguration operations to the user interface.

Simple.[Add, Duplicate, Delete] The class of *Simple* reconfiguration operations can be refined by the operations *Add*, *Duplicate*, and *Delete*. Applied to interaction logic, the operation *Add* adds a certain interaction process associated with a newly added interaction element that triggers the new interaction process in the interaction logic; *Duplicate* duplicates an existing interaction process in the interaction logic that is also associated with a new interaction element that has been added to the physical representation; *Delete* deletes a certain interaction process from the interaction logic and also deletes its associated interaction element as long as it is not associated with another interaction process.

Combination.[LooseCombination, Fusion, Partition] The class *Combination* describes a certain way in which reconfiguration operations combine interaction processes. Its three main subclasses are specified in Figure 4.13: *LooseCombination*, *Fusion*, and *Partition*. The sub class *Fusion* describes operations that, if applied to two or more interaction processes, fuse them into one process based on a certain fusion semantics. The following list introduces three types of fusion operation:

- *Parallelization*: When two or more interaction processes are parallelized, they are activated simultaneously. This process stops after the last subprocess has finished. Because all the subprocesses start at the same time, they also end in the same sequence depending on their individual time to be finished. Thus, the frequency of subprocesses finishing is implicitly defined. This could influence the semantics of parallelization such that the subprocesses are not 'only' parallelized.

- *Sequencing*: Sequenced interaction processes are activated in a well-defined sequence. After the first process in a sequence finishes, the next is activated, and so forth.

- *Timed Sequencing*: This operation is similar to the sequencing operation with the exception that the interaction processes succeeding each other are delayed by a specified wait time.

Partition is the inverse of the fusion operation. It divides one interaction process into two or more. Partition of an interaction process is not simple and should be investigated further. Still, it seems necessary to develop various heuristics for partitioning interaction processes paired with user intervention. *LooseCombination* is a class of reconfiguration operations defining all types of combination interaction processes that are not *Fusion*. A possible example is the combination of two interaction processes that process inputted text, each in a different way. These processes can be combined in a conditioned way, which means that if a certain input is to be generated on the physical layer, one or the other process will be activated. All reconfiguration operations

in the class *Combination* are combined by adding new interaction elements to the physical layer of the user interface, triggering the new, combined interaction process.

Discretization.[Single, Multiple] The class *Discretization* of interaction processes refers mainly to changing an interaction process awaiting a continuous value into an interaction process awaiting one specific, discrete value (*Single*) or a set of discrete values (*Multiple*). What is meant by continuous and discrete values in this context? An interaction element that generates a continuous value is an interaction element that generates one value out of a specific range of values. Therefore, as a minimum, any associated interaction process has to handle all the values from this range. A discrete interaction element generates only one specific value, for instance, a button generating a press event object. Applying a discretization operation to a particular interaction process results in a interaction process that transfers a continuous input value into one (*Single*) or one of a set of values (*Multiple*) out of the range of the accepted values for the interaction process. This kind of reconfiguration is normally associated with adding a new interaction element to a physical representation or replacing a continuous interaction element with a discrete one.

Influence.[Exclusion, ConditionedInfluence] In addition to combining or extending interaction processes, it is possible to extend interaction logic by defining the influences of and between interaction elements through their interaction processes. Here, the data processing is not changed, but the type of interaction between different interaction elements (not processes) is modified. *Exclusion* and *ConditionedInfluence* are only two of many specific examples of influences between interaction elements. Therefore, *Exclusion* can be subdivided into the following subclasses.

- *Direct Exclusion*: Applying *Direct Exclusion* to an interaction elements' interaction process adds further dependencies between the interaction elements (through their interaction processes). Take, for example, a button that is disabled until a certain number of characters have been entered. Thus, the button's disabled state depends directly on the information entered into a second interaction element.
- *Indirect Exclusion*: This kind of reconfiguration operation takes into account mainly those influences that result from data generated by the system. For instance, if a data value reaches a certain threshold, a button is enabled or disabled.
- *Timed Exclusion*: Influences between interaction elements can also be extended by adding time delays. This is true for both direct and indirect exclusion.

In the reconfiguration class *Influence*, no adaption of the physical representation is necessary.

Figure 4.14 shows three examples of the application of reconfiguration operations to an existing user interface on both levels: interaction logic and physical representation. The first example on the left side of Figure 4.14 shows the result of a *Parallelization* of two interaction processes by adding a new button to the physical representation labeled 'Input A+B' and a place/transition construction to the interaction logic indicated as dark gray circles (places), squares (transitions), and arcs. If a user generates an event by pressing the new button, the upper dark gray transition fires and adds a token to the downstream place. The following transition then fires and triggers both interaction processes associated with the buttons labeled 'Input A' and 'Input B'.

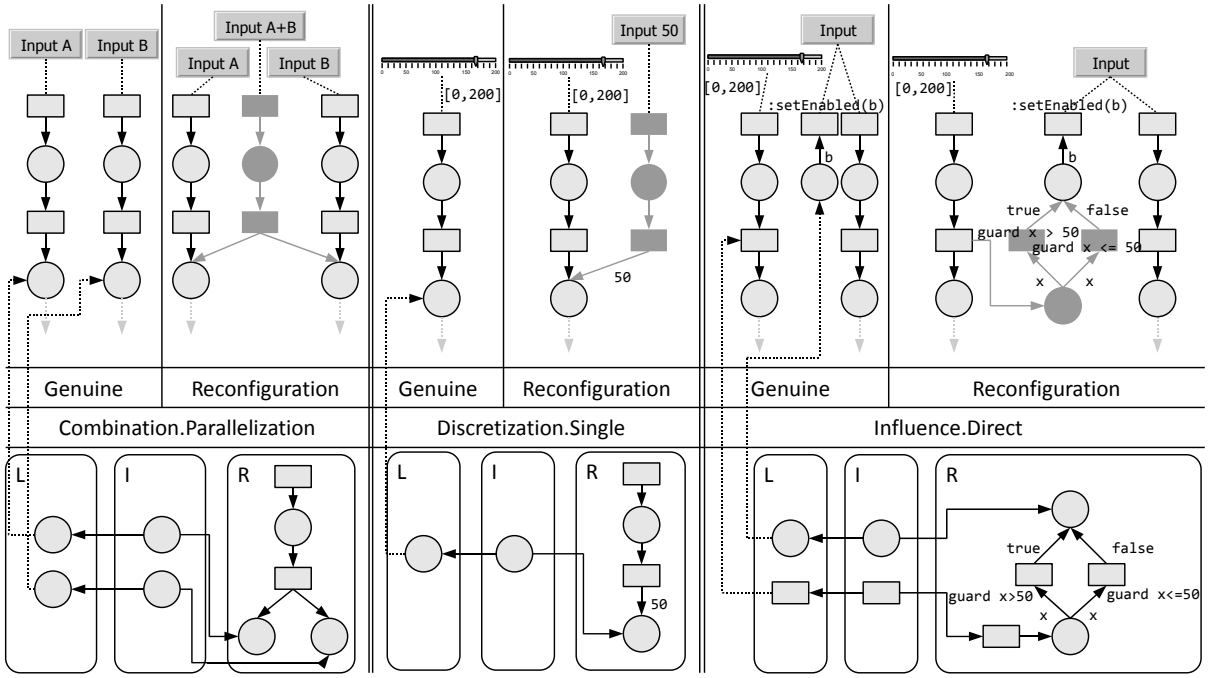


Figure 4.14.: Three examples for reconfiguration interaction processes for input data

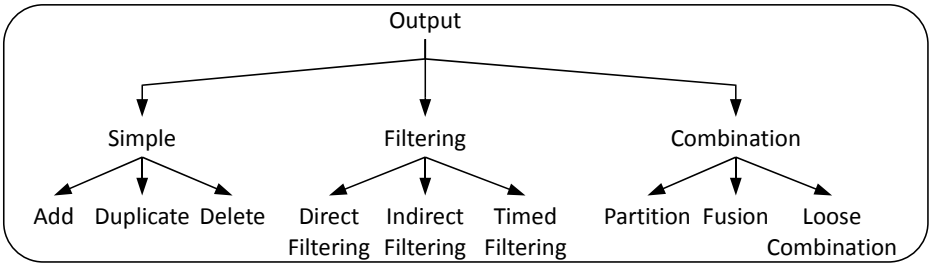


Figure 4.15.: Taxonomy of possible reconfigurations of formal interaction logic—Reconfiguration of output data processing interaction processes

The reconfiguration example in the middle of Figure 4.14 shows a *Discretization* initializing the original interaction process with the value 50. The example on the right side shows how certain influences between two interaction elements can be introduced to a given interaction logic. Here, the entered value of the slider influences the enable status of the button labeled `Input`.

In interaction processes that mainly process data outputted by the system (interface), other types of reconfiguration operations can be identified as described in the following paragraphs and shown in Figure 4.15.

Simple.[Add, Duplicate, Delete] Simple reconfigurations like *Add*, *Duplicate*, and *Delete* are defined similarly to those for interaction processes that process input data.

Filtering.[DirectFiltering, IndirectFiltering, TimedFiltering] By adding *Filtering* to an interaction process, data that is sent to the physical representation of a user interface for being presented to the user can be filtered in various ways. Thus, *DirectFiltering* means adding some interaction logic operation to the interaction process that applies specific data processing to

the sent data. *IndirectFiltering*, in contrast, involves adding other information to the filtering process, like different output data sent by the system to the interaction logic. By involving aspects of time, for instance, a delay in filtering output data, this reconfiguration can be called *TimedFiltering*.

Combination.[Fusion, Partition, LooseCombination] The *Combination* of output data replaces a simple—for instance, one dimensional—interaction element, which only outputs a single value, with an interaction element that presents two, three, or more values in relation to one another (for instance, as a graph). In the above-described sense, this *Combination* can also be called *Fusion*. A *LooseCombination* is a less strict fusion of output values that combines the values and sends them to an output interaction element without necessarily arranging them in a relationship with one another.

Figure 4.16 provides two examples showing the application of *Fusion* and *DirectFiltering* to interaction processes for output data. It is noticeable that *Fusion* is paired with a change of physical representation, unlike *DirectFiltering* in which only data processing is changed on interaction logic level.

As mentioned above, this list of reconfigurations of interaction processes for input and output data is neither complete nor self-contained. For instance, *Influence* reconfigurations for interaction processes for processing input data can also be applied to interaction processes for processing output data, and so forth. The goal was to identify certain reconfiguration operations or groups of operations that can be seen as a basis for more specific operations that are often highly dependent on their context. For applying these reconfiguration operations to a given interaction logic, the following section will introduce some basic requirements and ideas of how reconfiguration can be applied to a given interaction logic.

Applying Reconfiguration

The application of reconfiguration to formally modeled user interfaces must be examined from various perspectives. Two perspectives are those of the participants in the reconfiguration: (a) the user and (b) the reconfiguration system. Further instances that could trigger reconfiguration operations can be super-users or supervisors of a given controlling scenario or other processes that intervene. One might also discuss to what extent triggering reconfiguration and the reconfiguration itself can be directly modeled into formal interaction logic. This requires further development of the reconfiguration system and its implementation and will be the subject of future work.

A further perspective considers at which point in time a reconfiguration should be applied to a user interface. Another is that of the data involved in generating reconfiguration operations before applying them to a user interface. The following sections will describe the various perspectives, discuss implementation, and indicate areas where further investigation is needed.

Triggering Reconfiguration

Triggering a reconfiguration at a certain point in time can be defined as a time-dependent event that describes the application of a reconfiguration operation to a given user interface. This trigger event can be caused by the user or by a reconfiguration system. There are various possible objectives for initiating such an event. For example, the reconfiguration system may identify a failure in the interaction between user and system. On the other hand, the user may

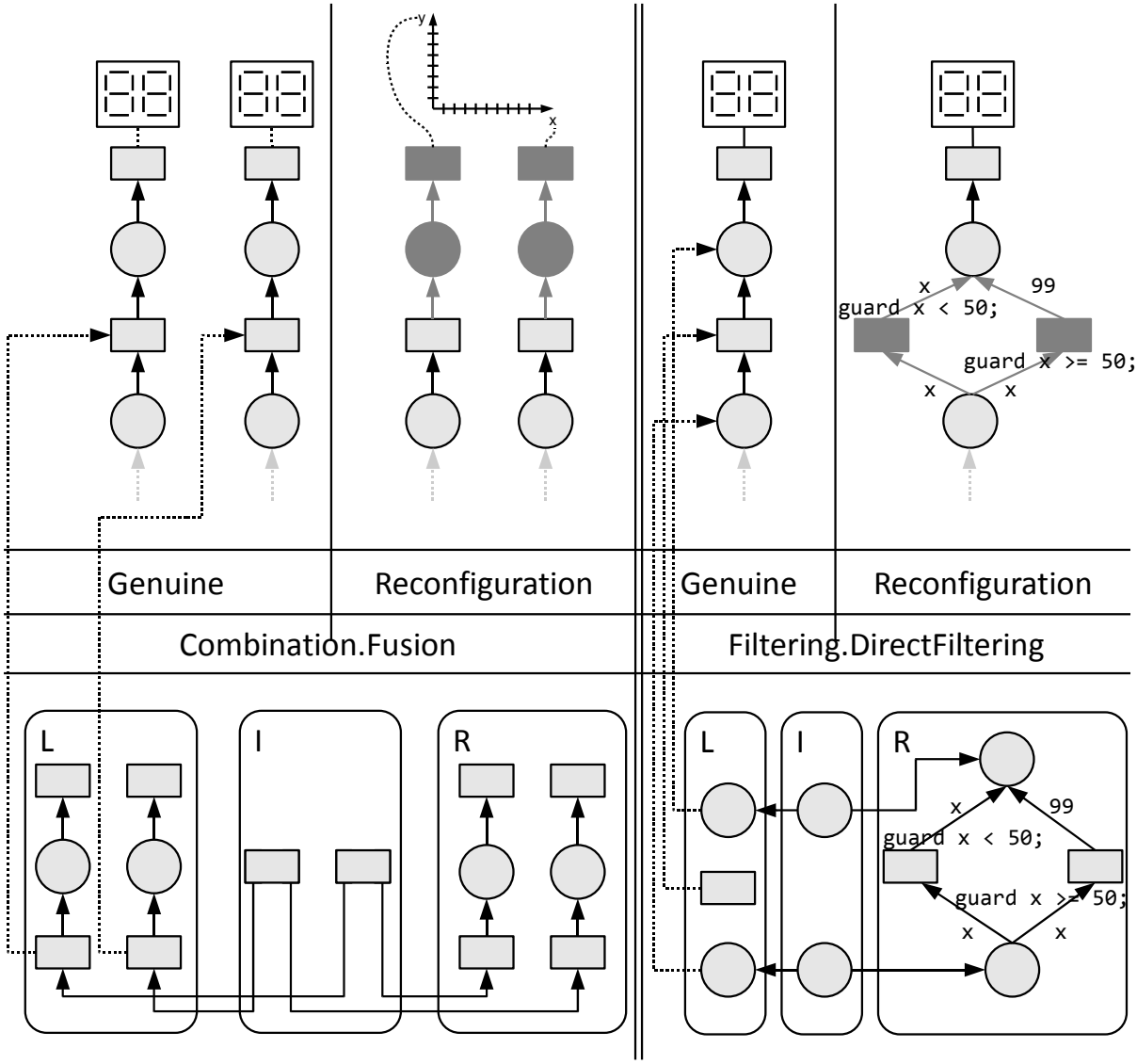


Figure 4.16.: Two examples of reconfiguration interaction processes for output data

want to change the behavior of the user interface because, for instance, the initial design is not best suited for solving a particular task. Schubert [245] describes a software system that controls the triggering and application of reconfiguration and redesign for a given user interface in the context of learning systems. To give a brief overview of possible reasons for triggering such a reconfiguration in other contexts, the following list mentions some scenarios:

Adaption to user's understanding and expectation Here, the user triggers a reconfiguration if the user interface does not behave as expected. A simple example is this: if a user presses a button and the reaction is not the expected one, that user would adapt the interaction process connected to the button.

Erroneous interaction Detected errors in interaction can trigger reconfiguration of a user interface designed to prevent such errors during future use. This kind of triggering and application of reconfiguration of the user interface is currently under investigation.

Automation and supervision Applying reconfiguration to add more automation and system-side supervision to system controls and interaction logic can serve various purposes. A simple one is that of meeting the changing requirements on a user interface issued by a local authority, a company, or some other potential user.

Practice and training of the user Practice and training in the use of a given user interface can lead to user reconfiguration of the interface. For instance, after becoming familiar with how outputted information is presented by the interface, the user may choose to improve that presentation.

There are many other possible scenarios that can cause reconfiguration operations to be triggered. Still, all of them are influenced by the following factors:

The user, who shows a certain amount of knowledge concerning the task, the system, and the user interface, as well as various cognitive abilities.

The system, which can identify and manipulate the triggering of reconfiguration in various ways—whether directly by applying certain changes to interaction logic or indirectly through the user.

The environment and situation, which influence both the user and the system in various ways.

The user interface offering various channels for interaction.

Thus, triggering of reconfiguration involves a broad range of considerations to be investigated in future work. Still, some basic work has already been done in Schubert's thesis [245] that should be transferred to general purposes and to a system that offers instantiation of the above-introduced concepts of *user* or the *group of users*, the *system*, the *environment*, and, nevertheless, the *user interface* and it supported modalities.

Manual vs. Automatic Reconfiguration

This aspect of applying reconfiguration to a given user interface mainly aims at the type of generating concrete instances of classes of reconfiguration operations described in the former section. Here, in a next step after triggering a reconfiguration, the used information and data is of main interest that has to be used to generate specific reconfiguration. These types of information and data needed for generating reconfiguration operations can be associated to three different groups on how reconfiguration is applied to a given user interface:

Manual reconfiguration is mainly based on the user's need to adapt a given user interface. Here, the user decides how to reconfigure the user interface and which parts of the original user interface are involved in the reconfiguration.

Automatic reconfiguration Here, a reconfiguration system is added to a framework that applies reconfiguration to a formally modeled user interface. The system can identify certain situations during runtime in which reconfiguring a user interface can make it perform its functions more safely, more efficiently, more automatic, and so forth. For instance, if it is possible to identify failures in interaction, the reconfiguration system could generate a reconfiguration to ensure that this failure cannot recur.

Semi-automatic reconfiguration This is a combination of both manual and automatic reconfiguration. For instance, the user identifies problems in using the system during interaction but does not know how to correct this issue using reconfiguration. Here, a reconfiguration system based on various heuristics and algorithms could generate proposals for reconfiguration of the user interface. The reverse might also apply; the reconfiguration system could identify failures and propose certain reconfigurations to the user, but not directly apply them to the user interface.

An automatic approach to identifying situations in which a reconfiguration should be applied to an interaction logic and determining what kind of reconfiguration would be appropriate in a given situation will be the subject of future investigation. First of all, it will be important to ascertain when to reconfigure a given interaction logic and user interface, how the reconfiguration should be carried out, and who should do it. These issues will be further discussed and an approach described in Chapter 8.

Creation of Reconfiguration Rules

The creation of reconfiguration rules involves various types of information. First, it is necessary to know the type of reconfiguration to be carried out. Without this knowledge, it is not possible to create the necessary DPO rules since the type of reconfiguration determines which the rule-creation algorithm will be used. That is to say, for every type of reconfiguration, a specific algorithm has to be developed and implemented.

After the correct algorithm for the reconfiguration operation has been selected, the interaction processes that will be affected by the reconfiguration have to be identified and the specific parts of them derived using graph algorithms. For this purpose, various approaches may be used. For instance, for an approach where the user selects certain interaction elements to represent the related interaction processes, the algorithm has to identify the parts of the interaction logic that are connected with these interaction elements and thus identify the interaction processes involved.

Now, the selected algorithm for creating a DPO rule from the derived interaction processes has to create the left and right side of the rule and the interface graph, which are all connected via the matching functions. Based on the XML-based data structure, the algorithm generates the output as three graphs in PNML format and a mapping node that associates the three graphs with each other by the id of places, transitions, and edges. Several parts of this creation workflow for DPO rules will be described in the next chapter, which introduces the framework implemented developed in this dissertation to create, simulate, and reconfigure formal user interfaces.

As shown in Section 4.2, most reconfiguration operations also apply changes to the physical representation of a user interface. In the context of reconfiguration, differentiation between the physical representation and the interaction logic is often not as strict as described above. Still, if the term reconfiguration is used, it usually refers to changes in the interaction logic rather than to changes in the physical representation, with the exceptions of adding, deleting, or replacing interaction elements. In the following section, more complex operations for redesigning the physical representation of a user interface will be described.

4.3. Redesign

Redesign is the change of the physical representation PR of a user interface, which was defined as a tuple $PR = (I, P, C, p, c)$ in Section 3.2. The formalization of the redesign can thus be defined as follows:

Definition 4.15 (Formal Redesign) *A given physical representation*

$$PR = (I, P, C, p, c)$$

is redesigned to a physical representation $PR' = (I', P, C, p', c')$*, if* I *is replaced by* I' *and if function* p *is replaced by a function* $p' : I' \rightarrow P^*$*, where*

$$\exists i \in I' : p'(i) \neq p(i),$$

and function c *by an adequate function* $c' : I' \rightarrow C$*.* ◇

Definition 4.15 defines the redesign of a physical representation of a given user interface as the replacement of set I with another set I' of interaction elements and the replacement of function p with a function p' that associates any interaction element of the new set I' of interaction elements with specific physical parameters. Function c has to be replaced by function c' only because of the replaced set I' of interaction elements. Furthermore, sets P and C are assumed to be generally valid and complete concerning all possible physical parameters and classes of interaction elements.

This approach to redesign is very general. As introduced in Section 3.6, there are various XML-based description languages for modeling user interfaces, especially their outward appearance. To transfer the formal approach to user interface redesign described above, XML-based technologies have to be applied to the description of the physical representation. This kind of formally modeled redesign can be applied using transformation technologies like XSLT [275]. Based on these technologies, redesign rules can be also modeled formally as XSLT schemas or other kind of XML rewriting approaches. The use of the implemented graph rewriting approach seems not to be suitable in this context because physical representation is not a graph-based structure. For this dissertation, the redesign of user interfaces was implemented based on visual and interactive editors as will be described in Chapter 5, and therefore, XSLT-based approaches are not of interest at this point.

In context of multi-view physical representations as defined in Definition 3.41, an extension of the upper given definition of formal redesign is simple. The approach described has only to be applied to every view on its own.

In the formal redesign of the physical representation, adapting the 'behavior' of an interaction element is also of interest. Based on the formal approach sketched in Section 3.6, this process is based on the use of reference nets; techniques based on graph rewriting can also be applied here. Therefore, an implementation of formal redesign offers access to the formally modeled logic of an interaction element, as well as a handy interface for defining reconfiguration operations that are applied to the interaction element's logic using the same engine as the reconfiguration of interaction logic. This will be subject of future work on implementation and can be considered an extension of the UIEditor framework to be described in the next chapter.

4.4. Conclusion

This chapter introduced the reconfiguration and redesign of user interfaces as a formal approach based on the results of Chapter 3. First, the terms *reconfiguration* and *redesign* were defined to offer a solid nomenclature for this dissertation. In this context, it is especially important to differentiate between the modification of interaction logic and the modification of the physical representation of a user interface. After defining reconfiguration for the modification of the interaction logic and redesign for the modification of the physical representation, the ongoing discussion described formalizations for reconfiguring interaction logic based on the transformation of FILL models to reference nets as described above. For this purpose, graph rewriting systems were identified as a suitable formal approach for modeling reconfiguration. Ehrig et al. [73, 74] introduced the DPO approach for general graph rewriting and then adapted this approach to P/T nets, which are a simple type of Petri net formalism. Based on this foundation, Stückrath [265], together with the author of this dissertation, extended this approach to colored nets using an XML-based description language. To complete the formal approach to rewriting colored nets, the author further extends Ehrig's formalization.

After describing formal reconfiguration, the concrete application of reconfiguration to a given interaction logic was described by introducing taxonomies for various reconfiguration operations. These operations were categorized into those which process input data and those which process output data. In addition, the problem of triggering and applying reconfiguration was briefly discussed and identified as the subject of future work.

Finally, a formal approach to redesign was briefly discussed and the idea of the use of transformation languages for application to XML-based description languages was described. In context of this dissertation and the implementation that will be introduced in the next chapter, redesign is mainly implemented and applied by visual editors and by the user, who enters and changes the characteristics of the interaction elements of a given physical representation. Further investigation into how adaptive user interfaces can benefit from formal redesign remains to be conducted, which will also impact the UIEditor framework to be presented in the next chapter.

5. Modeling and Simulation of Formal User Interfaces

This chapter describes an implemented framework, including a visual editor for modeling the physical representation and the associated interaction logic using VFILL. Various modules will be examined that implement a simulation engine for a formally described user interface and a component for applying reconfiguration operations to it. The concrete implementation (Section 5.2) will be described on the basis of the architecture introduced in Section 5.1, which models the relationship between the modules and their internal structures. The description of the implementation will also involve the introduction of an XML-based format for serializing formal user interfaces. The chapter will conclude by describing various application scenarios and a set of possible extensions for future implementations (Section 5.3).

5.1. Introduction

Formal and visual modeling languages are often accompanied by tools that provide ways to use them in various application scenarios. The UIEditor framework provides the tools implemented in this dissertation to support an efficient way to model, simulate, and reconfigure formally modeled user interfaces and thus support an appropriate tool set for using VFILL to formally model user interfaces. Furthermore, this implementation provides interfaces for extending usable interaction elements and adding new components, such as a component for analyzing interaction logic, which is a topic for future work. It also uses various libraries for visualizing and simulating reference nets (cf. Kummer [149]).

From this less specific set of tasks and tools for the framework to be developed, a list of more specific requirements can be derived:

1. Modeling requirements:
   a) The framework should provide visual tools for modeling VFILL in an interactive process paired with a visual editor for modeling an associated physical representation, resulting in a completely formal user interface.
   b) The framework should provide tools for extending the initially created interaction elements based on approaches described above (cf. Section 3.6).
   c) Interaction logic modeled as a VFILL graph should be transformed automatically into reference nets using the algorithm described in Section 3.4.
   d) The framework should support all necessary implementations for easy editing of interaction elements and operations in VFILL.
2. Simulation and Reconfiguration:

a) The framework should support a module for simulating the formal user interface as created using the visual editors for modeling the interaction logic and physical representation. Thus, it should include the simulation engine implemented by Renew for simulating reference nets and combining them with the Java-based implementation of the physical representation as a Swing interface. Furthermore, it should implement a simulation engine that visualizes the physical representation and handles the event redirection from and to the simulated reference net representing the interaction logic of the simulated user interface.

b) The framework should implement a module for formal and XML-based reconfiguration of a given interaction logic using a visual editor for user-driven, interactive reconfiguration. Therefore, the reconfiguration module should implement a set of algorithms for such purposes as extracting certain interaction processes from the interaction logic. Furthermore, it should provide well defined and documented interfaces for adding automatic and semi-automatic modules for reconfiguration based on intelligent approaches.

3. Architecture and Implementation:

a) The framework should be implemented in Java to allow the use of Swing for visualization and event handling, Java reflection for easy addition of code during runtime, and object-oriented modeling of the framework to offer highly flexible extension.

b) The framework should implement interfaces for adding modules that provide various analysis methods. Here, various approaches for analyzing the interaction logic using methods known from Petri net analysis, as well as methods for investigating the interaction itself will be implemented in the course of future work.

c) The framework should support an interface for serialization and a set of serialization modules for importing and exporting different data formats. An XML-based format should be developed and modules for reading and writing it should be implemented, especially to make formally modeled user interfaces persistent.

As will be introduced below, these requirements result mainly from the software engineering approaches described by Rosson and Carrol [238]. Here, the authors introduce development processes for creating user interfaces as iterative processes involving methods such as scenario-based development for usability engineering. Such processes are based primarily on classic approaches familiar from general software engineering as described by Sommerville [257] with a focus on usability.

Figure 5.1 shows a more general process for creating user interfaces developed by Deming [57] and known as the *Deming Cycle*. The Deming Cycle consists of four phases: (a) plan, (b) model and create (do), (c) check and revise, and (d) act and deliver. The UIEditor framework supports all four steps in this process through its various modules. In planning, the UIEditor framework can be used by user interface modelers for creating mock-ups and testing them directly. During testing, a certain amount of reconfiguration can be applied to the user interface directly without the need to go back to modeling mode, which speeds up planning. In the next phase, the same group of persons is able to create a final model of the user interface by using the modeling tool in the UIEditor framework and, after completing it, to pass it directly to the users who will be testing it. Here, in the third phase (check and revise), the persons doing the testing, who are not necessarily user interface modelers, are able to use the interface to interact with the system

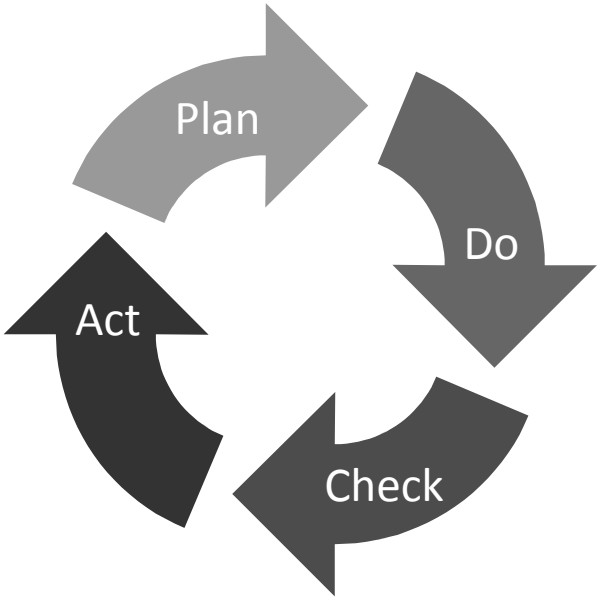


Figure 5.1.: Original Deming Cycle

to be controlled and, furthermore, to adapt the user interface to their own needs using the simple visual reconfiguration model. From the use and reconfiguration process, information can be generated to enhance the usability of the final user interface for delivery. In the last phase of delivery, the user interface will be used by the end user to control the real system. Furthermore, the end user will be able to reconfigure the user interface, where this data can be again used for a new planning phase, thus closing the development cycle. It can be seen that the formal approach supported by a solid implementation framework can speed up development and easily introduce the tester or end user to the development process without their having to deal with the problems that would result from a gap between evaluation data and its transformation to the concrete implementation of the user interface.

The following section will give a general overview of the architecture of the implemented framework that results from the above sets of requirements. This overview will be followed by detailed descriptions of the architecture modules.

5.2. The UIEditor Framework

The UIEditor framework is a framework for modeling, simulating and reconfiguring user interfaces based on Java programming language and extended by several libraries for XML parsing and visualization of graph-based data structures. Figure 5.2 shows the main structure and data communication between the single modules. In the illustration, arrows indicate data flow. Furthermore, as can be seen in the diagram, the UIEditor framework is divided into various modules, which themselves are further divided into various interrelated components. For data communication, various types of data formats were implemented, also indicated in Figure 5.2. Thus, every defined data format structures data communication between modules in the UIEditor framework and can be also associated with a serialized form. If there is no data format specified, data communication is implemented via object-based communication on the implementation level using object-oriented data-structures.

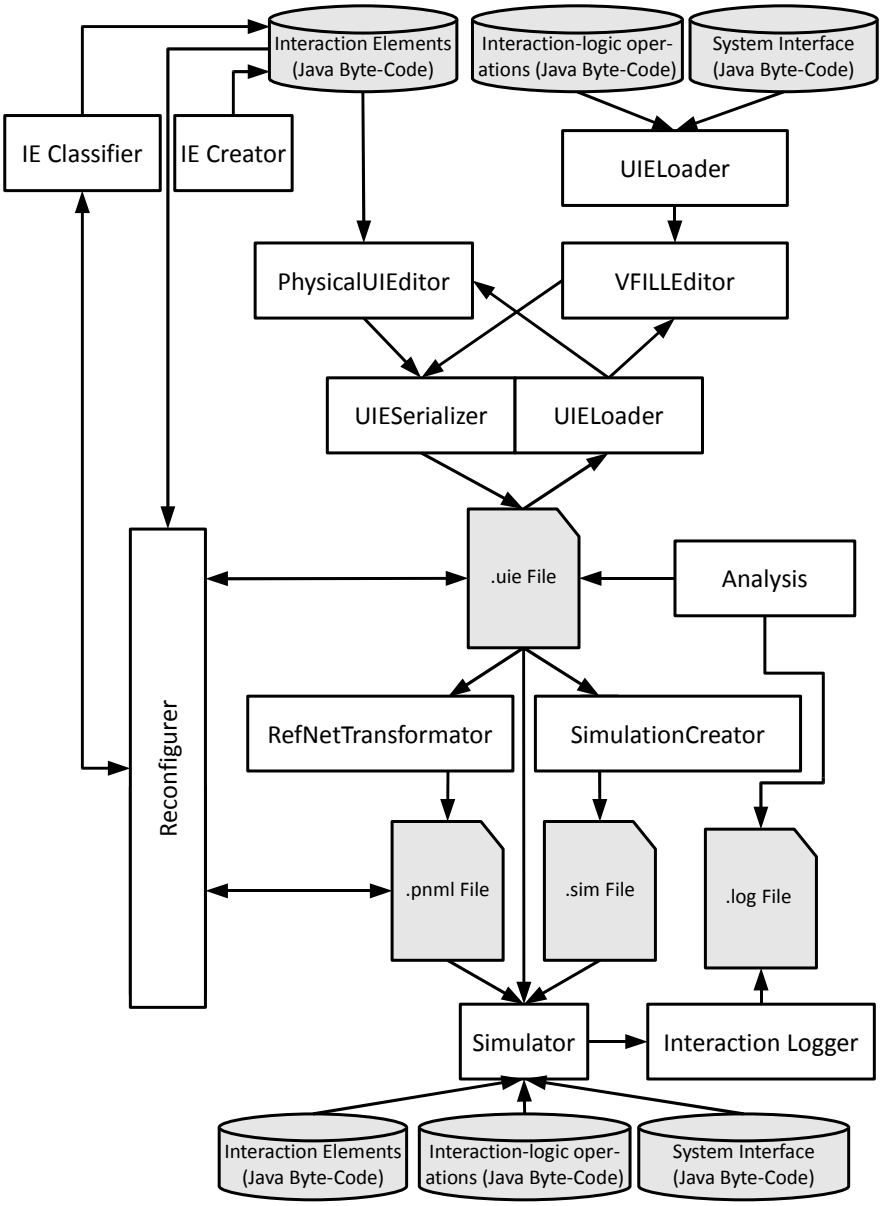


Figure 5.2.: Diagram showing the structure of the modules of the UIEditor framework

The following briefly introduces the modules. Thereafter, specific parts of these modules will be described in detail.

PhysicalUIEditor This module implements all necessary elements for interactive and visual modeling of the physical representation of a user interface based on the drag-and-drop metaphor. The modeling process is based on the 'what you see is what you get' concept familiar from graph drawing tools such as Open Office Draw[1], Microsoft Visio[2], etc.

VFILLEditor This editor provides a canvas for modeling VFILL graphs based on the same concept, look, and feel as the PhysicalUIEditor. Operations and BPMN nodes are

[1] http://de.openoffice.org/product/draw.html
[2] http://office.microsoft.com/de-de/visio/

added to the canvas via drag and drop after selection from a palette. Proxies are automatically added by the editor to the graph by associating it with a selected interaction element; channel operations can be added by using buttons from the icon bar at the top of the editor. Furthermore, by loading Java classes via Java Reflection, the implemented interface classes for connecting the system (System Interface in Figure 5.2) and the implemented interaction-logic operations (Interaction-logic operations in Figure 5.2) to the modeled user interface automatically generate system and interaction-logic operations in the VFILLEditor as visually represented operation boxes.

Simulator The simulator connects all data files and implementations of interaction logic operations and the system. It loads Renew to simulate the reference net and to handle the communication between Renew, the rendered physical representation, and the system's implementation. It also logs interaction during simulation runtime, which is of particular interest for further investigation of the automatic generation of reconfiguration rules based on analysis of interaction. For this purpose, a module called *Interaction Logger* has been implemented, which is closely connected to the simulator, which streams the applied interaction operations to the logger component.

Reconfigurer The module for user interface reconfiguration combines various software elements in one module. The first element is a interactive editor-like user interface, implementing interactive reconfiguration through the physical representation of the user interface. The second element is the implementation of a Petri net rewriting engine that has been implemented to apply DPO rules (cf. Section 4.2) to colored Petri or reference nets. The last element is a component that automatically generates DPO rules from the inputs the user applies to the psychical representation paired with application of reconfiguration operations to his selections. It also implements algorithms that extract affected sub-nets from interaction logic to create the correct matching functions for the DPO rule.

UIESerializer Many components were developed for serialization tasks and were subsumed in the serialization module. For instance, the UIESerializer serializes the modeled user interface in a specific XML-based format. This format has been developed especially to make these models persistent and enable their transformation, for instance, into reference nets as PNML files.

UIELoader In addition to serialization for defining an XML-based format and making modeled user interfaces persistent, loading abilities for serialized user interfaces are of interest. Furthermore, the UIEditor framework should include the integration of given implementations of interaction-logic operations and system interfaces based on Java. To this end, various components were implemented that handle the loading of user interfaces for modeling, simulation, and reconfiguration, as well as loading implementations of interaction-logic operations and system interfaces; these form connections to the system to be controlled using Java reflection. These implementations of interaction-logic operations and system interfaces have to be compiled as Java source and byte code for loading during runtime. For connectivity reasons, the UIEditor framework supports various Java interfaces for implementation and makes them accessible to the UIELoader.

Analysis The analysis module functions mainly as a placeholder for future work implementation. Still, this author has conducted various works on analysis of interaction logs.

Altunok [3] implemented different string and pattern matching algorithms for the identification of errors in interaction. In a current work by Borisov [29], an implementation will extend Altunok's work to a multi-user scenario, supported by a visualization engine for interpreting interrelations between multi-user interaction logs. Still, these are only a few examples of planned developments in the analysis of formal interaction logic and user interface modeling. In future work, verification and validation techniques will be used and implemented on the basis of the representation of interaction logic as reference nets to identify errors in the user interface and describe formal usability.

The following explanations are structurally organized according to a possible workflow in the creation, simulation, and reconfiguration of user interfaces using the UIEditor framework. Such a workflow will also be presented in the upcoming chapters (cf. Chapter 6 and 7), which describe studies evaluating the evolution of the active principals of building and using mental models on the part of human users interacting with machines. Figure 5.3 illustrates a basic workflow for using the UIEditor framework, where a user interface is first created in a classical software development process (here subsumed as *creation phase*) using the visual editors of the UIEditor framework. Next, the existing user interface is used in a *simulation phase* for testing or as a deliverable for use in controlling a given process. Next comes the *adaption phase*, where the user interface is adapted for various reasons using reconfiguration techniques. This reconfiguration can result in new requirements for a new development of a user interface or in an individualized user interface to be reused for simulation and/or for controlling processes.

This rather raw-grained representation of a possible workflow will be described in more detail below. There, the different steps of the workflow will be matched to the module structure of the UIEditor framework (as introduced in Figure 5.2) and the processing of the various implemented data structures and data formats.

Interactive Modeling

The VFILLEditor and the PhysicalUIEditor modeling component provide the interactive modeling for the two layers of formal user interfaces presented in Section 3.2. Both are WYSIWYG editors [231] implementing the drag-and-drop metaphor for adding and deleting elements. They are based on the same software architecture and use the same library for graph visualization, JGraph, that was first developed and introduced by Bagga and Heinz [13] and released under Open Source BSD license[3]. This library implements a graph visualization canvas supporting interactive graph modeling via point and click. The whole visualization engine is based on Java Swing [164], which is especially suitable for (a) modeling physical representation of user interfaces based on Java Swing components and (b) implementing new types of nodes using the comfortable Java Swing lightweight components, which are highly flexible. The latter is important for implementing VFILL nodes, like system, interaction-logic, or channel operations, as well as BPMN nodes. JGraph also provides a data structure based on Java's basic graph data structures, which are supported by the Java API[4].

Before describing the components implemented in the UIEditor framework for interactive modeling of user interfaces, it is useful to identify the relevant software modules in the framework (cf. Figure 5.2) and examine how interactive modeling is subsumed in the general workflow (cf. Figure 5.3). Figure 5.4 shows the module structure and indicates the sequence of usage of the

[3] http://www.opensource.org/licenses/BSD-2-Clause
[4] http://download.oracle.com/javase/6/docs/api/

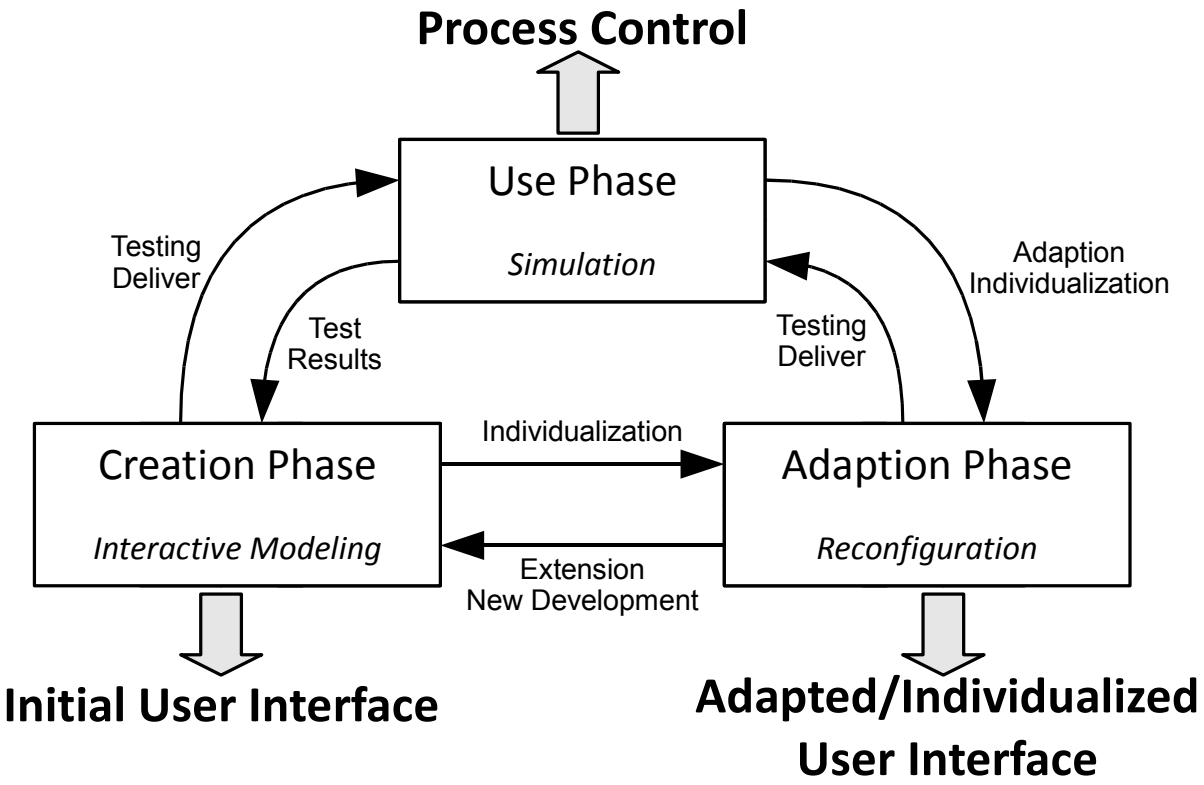


Figure 5.3.: Workflow for using the UIEditor framework to create, simulate, and reconfigure a formal user interface

modules needed for modeling user interfaces. Thus, in step ❶, interaction elements, the system interface, and interaction-logic operations are loaded from Java byte code stored as class files in the file system to the PhysicalUIEditor and VFILLEditor module using the UIELoader module. In step ❷, the user interface is modeled by adding interaction elements in the PhysicalUIEditor and defining specific interaction processes for these interaction elements in the VFILLEditor iteratively. An example of this will be described at the end of this section. After finishing the creation of the user interface, it will be serialized in step ❸ using the UIESerializer component to make the user interface persistent, on the one hand, and accessible and usable for other modules in the UIEditor framework, on the other (for such purposes as simulation or reconfiguration). A created user interface stored as a `.uie` file can also be loaded (step ❹) later on and thus reused in the iterative creation process for modification or extension.

Architecture and Data-Structure Model

The overlying architecture is a flexible, component-based implementation that provides easy extension for other implementations by more complex editors. Thus, the VFILLEditor and the PhysicalUIEditor are both implemented in the same container structure supported by the UIEditor framework. This architecture provides containers that can include palettes that perform functions like, holding interaction elements, as in the PhysicalUIEditor, and areas for positioning various types of canvases. For the PhyscialUIEditor, the class *JGraphCanvas* has been implemented by extending the basic visualization and editing canvases delivered by JGraph with drag and drop features. The VFILLEditor uses an extension of JGraph called *OperationCanvas*

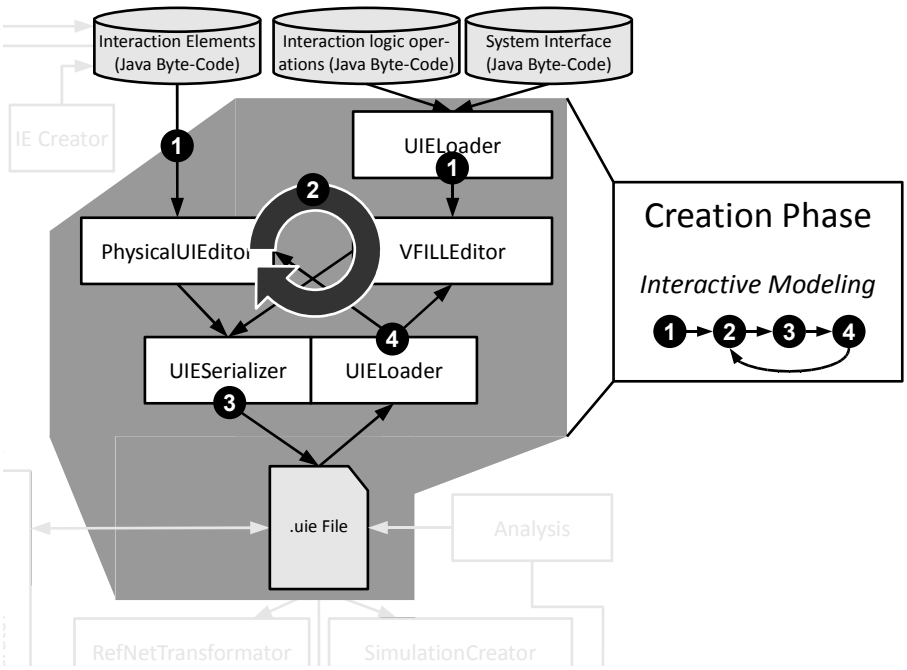


Figure 5.4.: Extraction of modules for interactive modeling of user interfaces using the UIEditor framework

with the same features as the JGraphCanvas, as well as other implementations, such as edge drawing, handling, and automatic layout of proxy nodes in the canvas.

The visualization is substantiated by a data structure that can be defined as a model in a Model View Controller architecture pattern described by Gamma et al. [94] and serves as the basis concept for implementation in the UIEditor framework. This complex data structure is shown in Figure 5.5 as a UML class diagram [195], which tends not to be complete but shows its main elements and classes. The main classes are the *Widget* and *Operation* classes, which represent, respectively, interaction elements and VFILL operations. Both are aggregated to *UIEditorGraphCell* objects to embed them on the JGraph architecture for visualization and editing. The *UserInterfaceModel* aggregates all the *Widget* and *Operation* objects of a modeled user interface in one object, which represents the modeled user interface. Any *Widget* object is associated with a *JGraph* object. This *JGraph* object represents the associated interaction process modeled in the VFILLEditor as a *JGraph* object. Here, the modeling convention described above which says that any interaction element is associated with one interaction process is implemented as a Java-based data structure.

Furthermore, a *Widget* object aggregates objects of type *WidgetInput* and *WidgetEvent*. These two classes represent input and output proxies as part of a *Widget* object. These objects are added to the aggregated *JGraph* object modeling the interaction process of a given *Widget*. Thus, *WidgetInput* and *WidgetEvent* are elements that are associated with a *Widget* object in a well-defined way and are also added to the associated interaction process of the given *Widget*. Therefore, *WidgetInput* and *WidgetEvent* are an adequate software implementation of the input and output proxies of a VFILL graph, which connect interaction elements to an interaction process.

Interaction elements modeled as *Widget* aggregate objects of type *WidgetParameter*, which define the physical parameters of a given *Widget*. A set of possible *WidgetParameter* objects

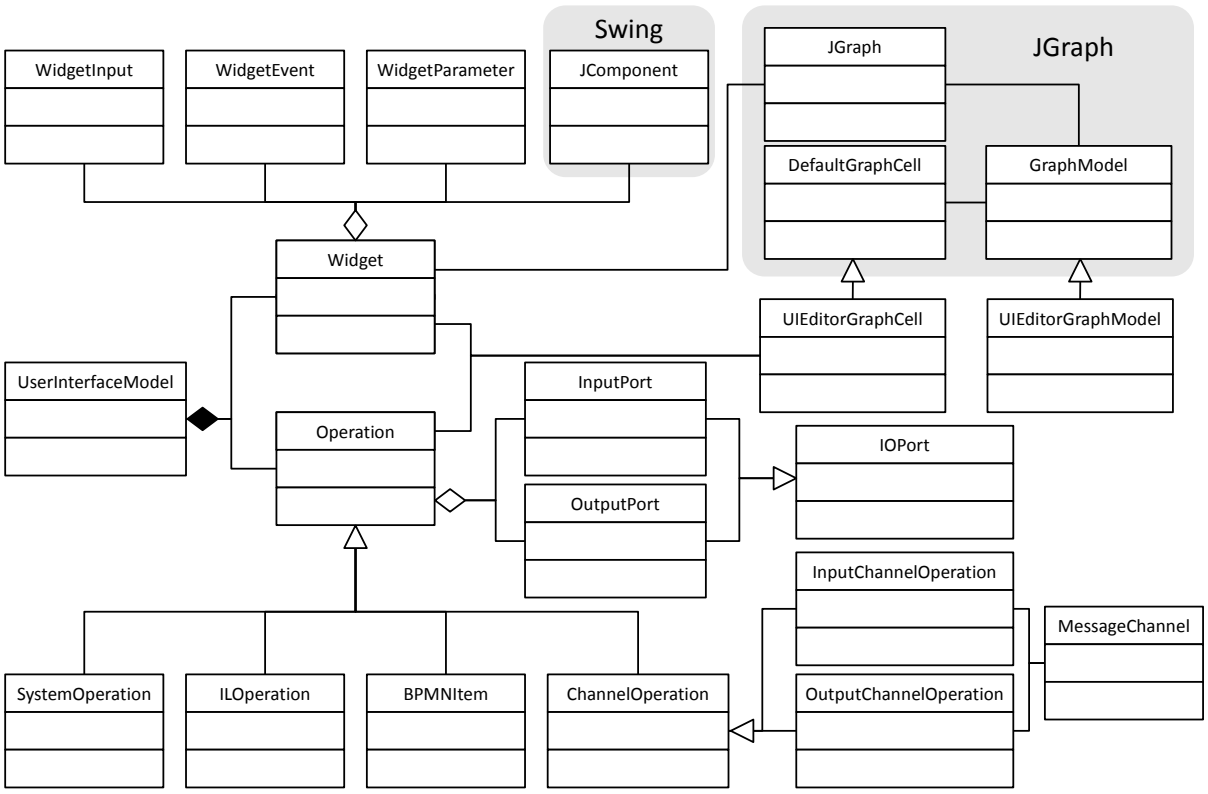


Figure 5.5.: Object-oriented data structure representing the formally modeled user interface during runtime and indicating connection points to other libraries

is defined in any subclass of *Widget* specialized to the specific *Widget* implementation, for instance a button. The outward appearance of a *Widget* is basically defined by the associated *JComponent*, which is the super class of every interaction element defined in the Java Swing library. For every specialized *Widget* class that defines the model of a widget, there is a specialized *JComponent* that specifies the view of the widget in the Model View Controller architecture pattern.

Interaction logic is modeled as set of operation nodes connected by edges. Therefore, FILL is modeled as set of classes extending the super class *Operation*, which is itself embedded in objects of type *UIEditorGraphCell*, such that they can be added to a *JGraph* object for interactive modeling and visualization. The data edges of a VFILL graph are modeled using the standard edge implemented in the JGraph library. Thus, the graphical structure of a FILL model is an object of type *GraphModel* and is part of the JGraph library.

Any operation aggregates a set of input and output ports implemented as class *InputPort* and *OutputPort*, which extend the class *IOPort*. Furthermore, class *IOPort* extends the class *Port* of the JGraph library, which cannot be seen in Figure 5.5. Objects of this class are aggregated in the *UIEditorGraphCell* objects and represent the points of connection for edges in a *JGraph* compatible with the definition of ports in VFILL.

Implementation of specific operations was realized by extending the class *Operation* with the classes *SystemOperation*, *ILOperation*, *BPMNItem*, and *ChannelOperation*. The *SystemOperation* and *ILOperation* classes represent Java methods in a VFILL graph. Thus, via reflection, the UIEditor framework implements a component that loads Java classes during runtime [167] and automatically creates instances of classes *SystemOperation* and *ILOperation*, depending on

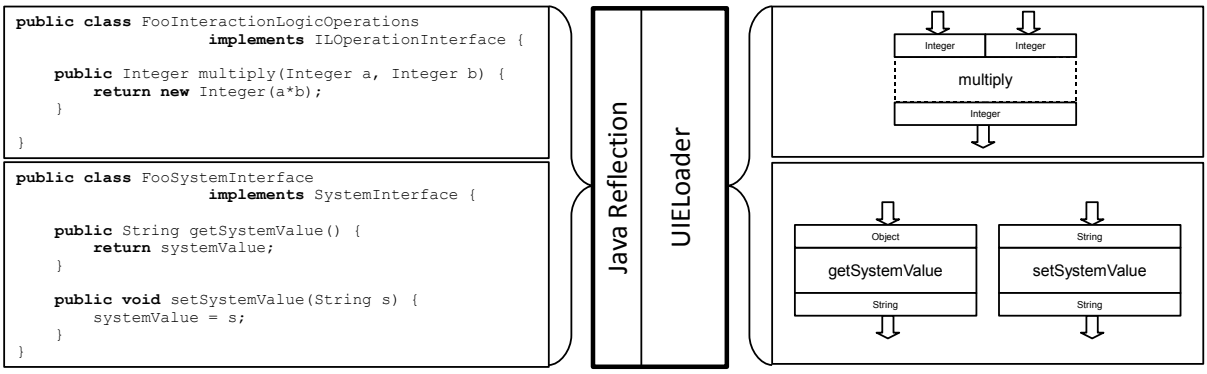


Figure 5.6.: Generating instances of classes *ILOperation* and *SystemOperation* using Java reflection API and a further component of the UIEditor framework

which kind of class has been loaded, which is defined by a certain implementation of a Java interface. The created instances then reference the method and retain this information for later serialization. For later simulation, the methods are loaded via reflection and are called by firing transitions in the reference net. An example of loading Java classes for instantiation of *ILOperation* and *SystemOperation* objects can be seen in Figure 5.6. In general, the UIEditor framework supports two Java interfaces: (a) The *ILOperationInterface* for implementing interaction logic operations as methods of an implementing class of the interface, and (b) the *SystemInterface* for implementation of a system interface for inputting and outputting data from the system. Thus, every public method of a class implementing the *ILOperationInterface* interface is interpreted as an interaction logic operation, where any parameter of the method is interpreted as an input port and a optional return parameter as an output port; any public method implemented in a class that implements the *SystemInterface* interface is interpreted as an input system operation (if there is a void return parameter and one method parameter) or as an output system operation (if there is a return and no parameter). Figure 5.6 shows an example of an implementation of *ILOperationInterface* and *SystemInterface*. The implementing class of the Java interface *ILOperationInterface*, called *FooInteractionLogicOperations*, implements a public method called `multiply`, which offers two parameters a and b of type Integer and also returns a value of type Integer. This method calculates only the product of the parameters a and b and returns the result $a * b$. The exemplary implementation of *SystemInterface* called *FooSystemInterface* implements two methods: `getSystemValue`, which returns a specific value, and `setSystemValue`, which assigns a value to a specific system value. These two methods are read in using the UIELoader, which creates two system operation boxes: an input system operation from the `setSystemValue` method, which offers one parameter and a void return value, and an output system operation derived from the `getSystemValue` method, which offers no parameters, but a return value of type String. More or less the same is true for the above-described example of an interaction logic operation with two parameters where the return value is not empty, resulting in the interaction-logic operation box that can be seen on the right in Figure 5.6.

Class *BPMNItem* represents BPMN nodes in the VFILL graph. BPMN items are not operations in the VFILL sense, but it is useful to implement them as extensions of class *Operation*. Given the necessity of including BPMN nodes in the visualization of a JGraph, it is easier to implement them in this way than to create them from scratch.

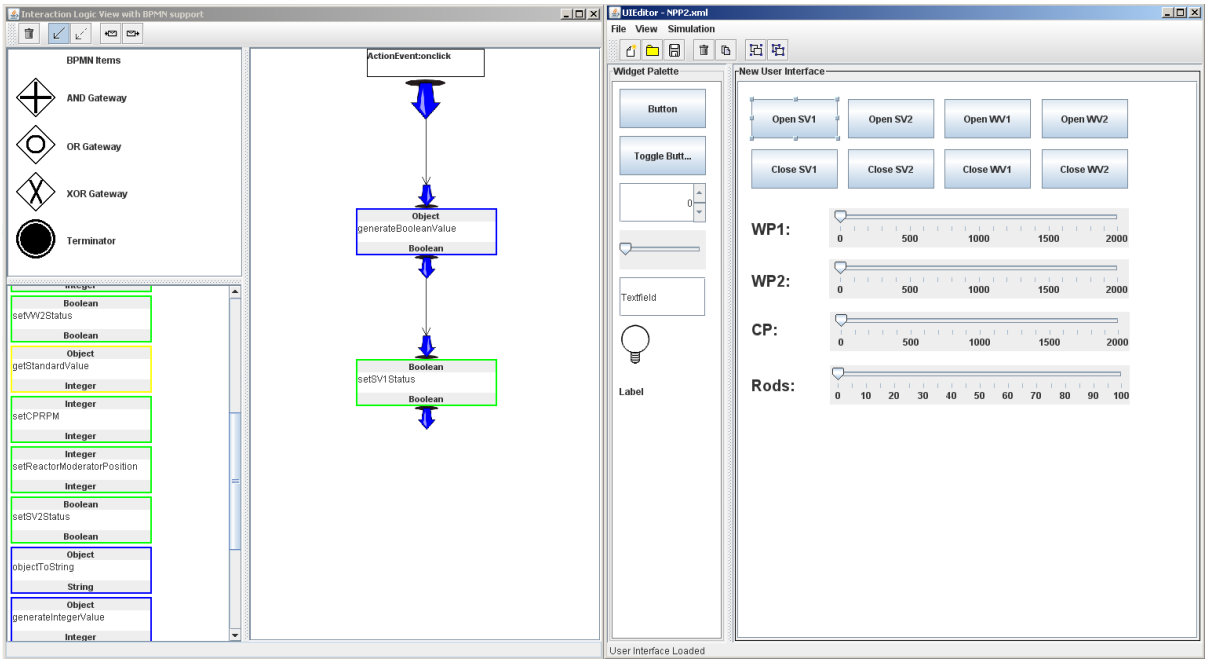


Figure 5.7.: Screenshot of the visual editors: On the left is the VFILLEditor; on the right is the PhyscialUIEditor

Class *ChannelOperation* represents channel operations and is extended by two sub-classes: *InputChannelOperation*, which defines the entrance of a channel, and *OutputChannelOperation*, which defines the exit of a channel. Every channel is modeled as an object of type *MessageChannel* and connects input and output channel operations as part of a *JGraph* graph model. The serialization of the entire data structure representing a VFILL graph will be examined in more detail below.

Visual Editors

The visual editors of the UIEditor framework are shown as a screen shot in Figure 5.7. The VFILLEditor can be seen on the left of Figure 5.7. The upper left palette provides BPMN nodes for adding them to the right side canvas (here shown with one proxy node and two operation nodes). The lower palette shows a set of operations, where the border's color indicates the type of the operation (system operation for input data, system operation for output data, and interaction logic operation). Operations can be added via loading Java classes like described before. Loaded interaction logic and system operations can then be added via drag and drop from the palette to the canvas for modeling a VFILL graph. Input and output channel operations are added by using the buttons in the toolbar at the upper part of the VFILLEditor. Furthermore, there is a button for deleting selected nodes. Another way to delete a node is by dragging it out of the canvas onto the operation palette. By dropping it there, it will be deleted together with its connected edges.

On the right side in Figure 5.7 is the PhysicalUIEditor. Its structure is comparable to that of the VFILLEditor. On the left in the PhysicalUIEditor, there is a palette of interaction elements; on the right is a canvas for positioning and adding interaction elements from the palette via drag and drop. The toolbar in the upper part of the editor provides operations for deleting or duplicating interaction elements, as well as opening, saving, or creating a new user interface.

The menu supports three menu items: (a) *File*, for saving, opening, and creating a new user interface or exporting it to other file formats, (b) *View*, for opening the VFILLEditor window, and (c) *Simulation*, for changing the mode from editing to simulation and for creating and loading simulation files.

To provide deeper insight into the modeling process using the visual editors of the UIEditor framework, Figure 5.8 shows an example of the process that can be used to model a user interface. In the first step (step ❶), a new physical element, here, a button, is added to the physical representation from the palette via drag and drop. In the next step (step ❷), the parameters of the interaction element are defined, here, the label. The label indicates that the newly added button opens something called 'SV1'. As it will be seen below, SV1 is a special valve in a simulated steam water circuit in the primary feed water circuit of a nuclear power plant.

By activating the button by clicking on it (step ❸), its associated interaction process pops up in the VFILLEditor, showing an canvas that initially contains only an output proxy representing the click event of the button element. Next, loaded system and interaction-logic operations can be added from the palette on the left to the canvas via drag and drop. Here, an interaction-logic operation called `createBooleanValue` is added; it creates the Boolean value `true`. Furthermore, a system operation has been added labeled `setSV1Status`; it sets the inputted Boolean value as the status of the valve SV1 (thus, true means 'open the valve SV1'). By connecting these two operations using edges, as shown in the last step (step ❺) in Figure 5.8, a click event of the button that occurs during runtime of simulation of the modeled user interface is sent to the interaction-logic operation, creating a `true` Boolean value. This value is then sent to the system operation, setting the status of valve SV1 to `true`. Now, pressing the button labeled 'Open SV1' sets the status of valve SV1 to `true`; this opens the valve as long as the created user interface is simulated with the associated system interface, the system, and the interaction-logic operation.

In the next section, the simulation module of the UIEditor framework will be introduced in detail, beginning with the creation of simulation files and ending with the loading and execution of a created simulation.

Simulation

The simulation module of the UIEditor framework provides components to create simulation files and to execute them. It does so by loading a modeled user interface, starting a connected system simulation, and connecting these two elements via the transformed reference net from the interaction logic modeled in the form of a VFILL graph. As part of the workflow shown in Figure 5.3, the simulation can be identified as a subsumption of modules in the UIEditor module structure shown in Figure 5.9. For the simulation, a `.sim` file referencing a specific user interface in the form of a `.uie` file and an interaction logic represented in serialized form as a `.pnml` file has to be created (step ❶). In step ❷, the simulation, which combines these files with all the interaction elements, implementations of interaction-logic operations in Java byte code, and the system interface in Java byte code, is loaded by the simulator. Then, simulation can begin. This is handled by the Simulator module, as is the interaction logger, which logs all interaction operations instigated by the user, as well as the system state as provided by the system interface's implementation (step ❸). In the following, these subprocesses will be described in greater detail, as will the relevant data structures and formats.

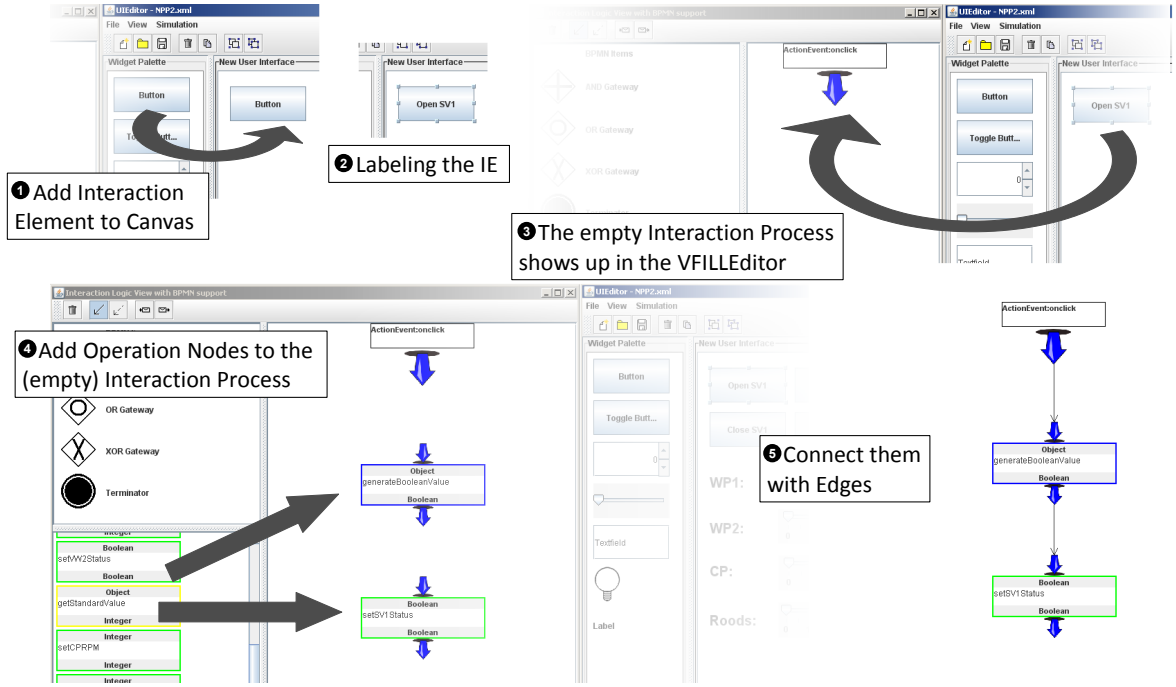


Figure 5.8.: Exemplary modeling process using the visual editors in the UIEditor framework to create the physical representation and interaction logic of a user interface

Creation of Simulation

Before starting and running a simulation, the simulation must be created as a file using the workflow that can be seen in Figure 5.10. The user of the UIEditor framework only has to select a certain user interface serialized as `.uie` file and then to define a name for the simulation file to be created. The rest of the workflow is implemented as a fully automatic creation process. The resulting file, which has the extension `.sim`, is a selection of references or URLs including one to the interaction logic in its representation as a reference net, a URL referencing the formal user interface, and URLs referencing the Java implementation of the system interface and the implementation of used interaction-logic operations. This reference net has to be created by the *RefNetTransformator* in PNML format [112] from the description of the user interface, a `.uie` file. The same is true for the stub file that is necessary for the simulation of the reference net, especially for its connection to Java code and the XML-based serialization of the user interface itself when using Renew. For the simulation, references to the implementation of the interaction logic operations and the system interface are also necessary. They are part of the user interface serialization. In conclusion, the *SimulationCreator* as part of the simulation module of the UIEditor framework consists of the *RefNetTransformator*, which generates a reference net from a given VFILL graph serialized in the `.uie` file using the introduced algorithm in Section 3.4, and the *RefNetStubCreator*, which generates a stub used by the Renew net simulator to apply communication between the Java code and the simulated reference net. The output is a simulation file (with the extension `.sim`) that references the stub, the reference net in PNML format, and the file representing the user interface with its physical representation as a set of Java Swing-based interaction elements, interaction logic in the form of a VFILL graph in its XML-based representation, and all references to the relevant *ILInterface* and *SystemInterface* implementations.

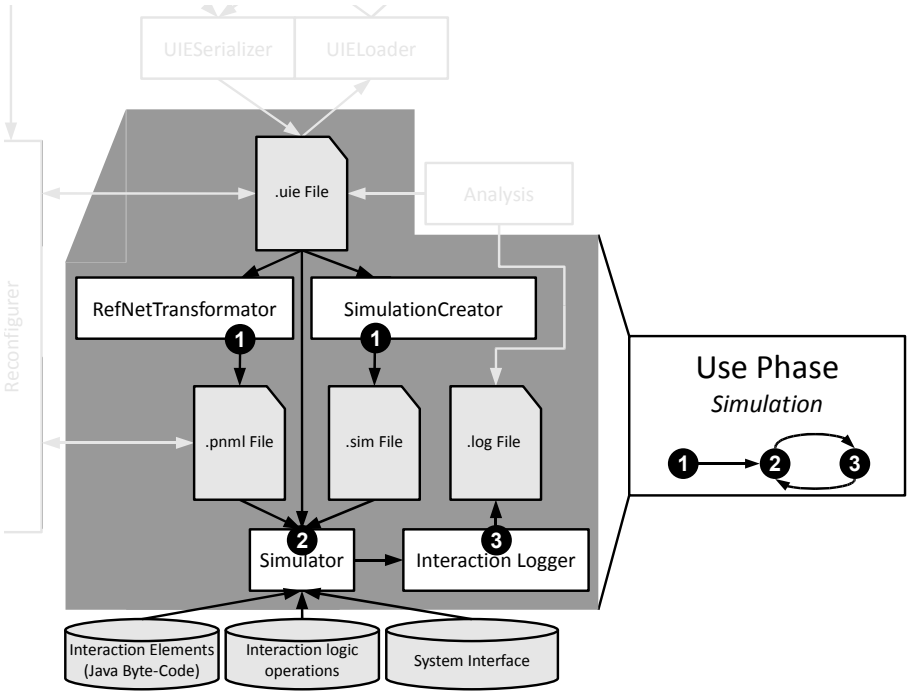


Figure 5.9.: Extraction of modules for simulation of user interfaces using the UIEditor framework

Loading a Simulation

The next step is to start and run a simulation. This process occurs when the simulation module loads a simulation file using the *SimulationLoader* component after a file has been selected by the user. Figure 5.11 shows the loading process. This process can be subdivided into three sub-processes that are related to the files referenced by a simulation file. In Figure 5.11, this file is named `foo.sim`.

The first process is handling the stub, which is necessary for the simulation of the reference net shown as file `foo.pnml` in Figure 5.11. The stub file is compiled by the *StubCompiler* in Renew, which creates a Java source file. This source has to be compiled by the *javac compiler*, which can be used via the Java package *com.sun.tools.javac.Main* during runtime. The resulting `.class` file is then read by the Java class loader, which is part of Java's virtual machine and the

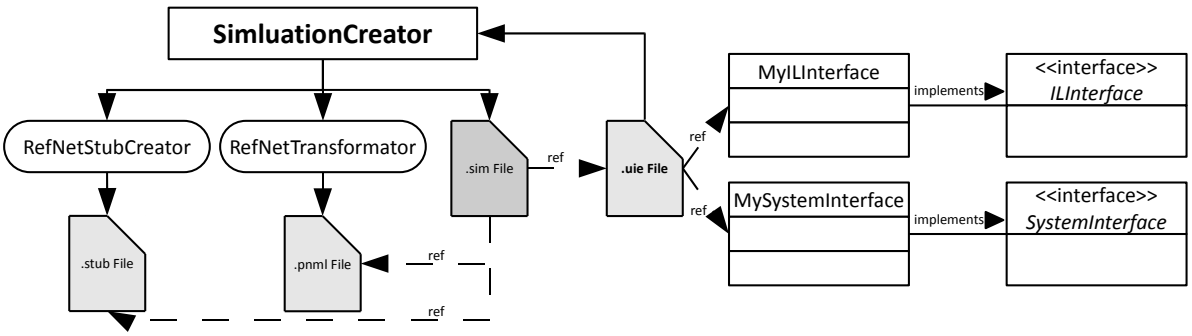


Figure 5.10.: Components and data objects involved in the creation of a simulation file with the file extension `.sim`

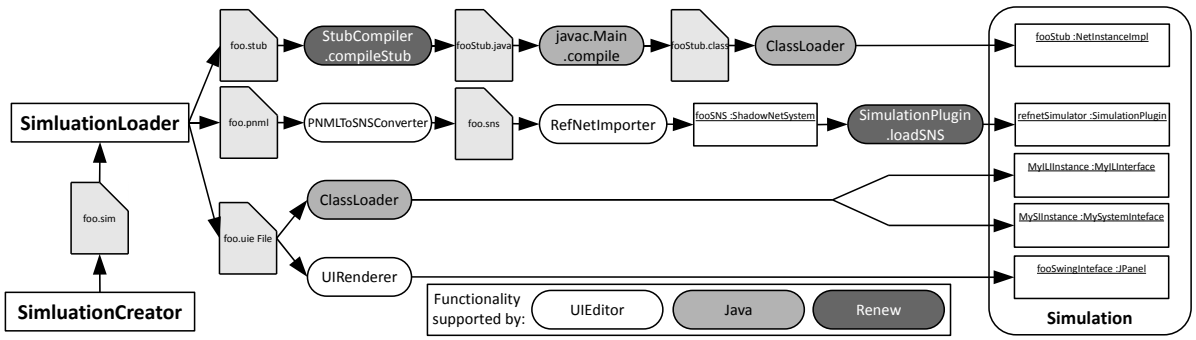


Figure 5.11.: Components and data objects that are involved in the loading process of a simulation file `foo.sim`

central module of the Java reflection API. Using the class loader, class files can be instantiated as objects of type *Class*. This object-based representation of Java classes during runtime makes it possible to reload classes. The resulting stub instance is the first part of the loaded simulation.

The second process in Figure 5.11 shows the instantiation of a reference net, shown as `.pnml` file; this is a shadow net system, a data structure defined and used by Renew. A shadow net system represents the reference net as a specific data structure meeting the requirements for simulation. For this purpose Jan Stückrath [265] implemented two software components: (a) the *PNMLToSNSConverter* (where SNS stands for Shadow Net System), which converts a reference net in PNML format to a serialized SNS form, and (b) the *RefNetImporter*, which instantiates a serialized SNS that can then be loaded by the *SimulationPlugin*, which is supported by Renew. The result of this process is Renew's *SimulationPlugin* initiated by a reference net representing the interaction logic of the given user interface.

The third and final process deals with the user interface itself. Using Java's class loader, the specified implementations of the *ILInterface* and *SystemInterface* interfaces are instantiated. The second part of the process generates a *JPanel* that contains all the interaction elements initiated, with the specified physical parameters that have been assigned to them using the PhysicalUIEditor.

All created instances will be accumulated in a data object of type *Simulation* for simpler transfer to the main component of the simulation module, the *Simulator*.

Running a Simulation

The last component of the simulation module described here is the simulator itself. Its main functionality is the organization of communication between the rendered physical representation as *JPanel*, the simulation of the interaction logic in Renew, and the system sending data from the net to the system interface and back from the system to the net.

In Figure 5.12, the *EventInterface* component can be seen as element for controlling the former described communication between the instances loaded by the *SimulationLoader*. The whole structure is multi-threaded. This means that all components, the physical representation as *JPanel*, the system and the interaction logic interfaces, and the simulation of the reference net in Renew are independent, only connected by passing data that is controlled by the *EventInterface*.

Communication by the *EventInterface* with the Swing-based physical representation is implemented using the Swing event system [164]. The *EventInterface* implements all existing listeners, such as the *MouseMotionListener* or the *ActionEventListener*. Depending on the

type of interaction element, the *EventInterface* will be added to it as a listener during initialization. Which listener is added to which interaction element can change, depending on the element. This information is separately stored in an XML-based configuration file. To identify the transition to be associated with a given interaction element, the *id* called *ieID* is used, which was introduced in Section 3.4. If data is to be sent for output to the physical representation, the correct matching of transition and interaction element is again implemented using the *ieID* and the `setMethod` appropriate to the interaction element. Data as to which method has to be called for each interaction element is stored in a separate file, which also defines the event mapping.

Communication to the simulated reference net in Renew is implemented using two components: (a) the stub, which handles calls to the net firing transitions, and (b) the implementation of the *EventInterface* as a singleton offering the possibility of defining methods as `public static`. This means that they can be called directly without containing an instance of the class *EventInterface*. The singleton pattern described by Gamma et al. [94] provides the ability to define a type that can only be instantiated once. This solves the problem of static access to methods and values of an object that can generate problematic side-effects in standard object-oriented programming. Here, using a singleton pattern makes it much simpler to call methods from the net because there is no need to pass an instance of the *EventInterface* class to Renew. Calls to the net are handled by the stub, where an instance can be accessed by the *EventInterface* without further problems.

For communication with the system and interaction logic interface(s), the *EventInterface* uses the same mechanism as used for communication with the physical representation. Here, *id*s are not used to identify the correct method (because there is only one method with any given name). Instead, they are used to identify the correct re-entry point into the net. For instance, if an interaction logic operation is triggered in the net, the net calls the associated method in the *ILInterface* implementation. After the method has finished its calculation and the result has to be sent back to the net, the correct re-entering transition is identified by an *id* through unification (cf. Section 3.4). This is necessary because one interaction logic operation may be used several times in an interaction logic. It would be fatal to fire the wrong re-entry transition because this would initiate the wrong interaction process and the wrong value might be set, to the wrong input value in the system. For system operations, there is no such problem because multiple appearances of a system operation are reduced in the transformed reference net to exactly one representation. For identification, the correct interaction process after a system operation has been finished, the *id* called *procID* is used to identify an interaction process.

If the simulator is prevented from simulating, the UIEditor framework will automatically load the reconfiguration module, which will described in the next section.

Reconfiguration

Figure 5.13 illustrates the extraction of modules by the UIEditor during the process of reconfiguration. Here, reconfiguration is sub-divided into three main steps, where step ❶ indicates loading of the `.uie` file, which represents the user interface to be reconfigured (mainly for redesign issues) and the `.pnml` file, which represents the interaction logic of the loaded user interface as a reference net (mainly for reconfiguring the interaction logic). After loading these two files along with the user interface, reconfiguration operations are iteratively applied to the user interface (step ❷), which sometimes involves the *IEClassifier*, which supports the user in

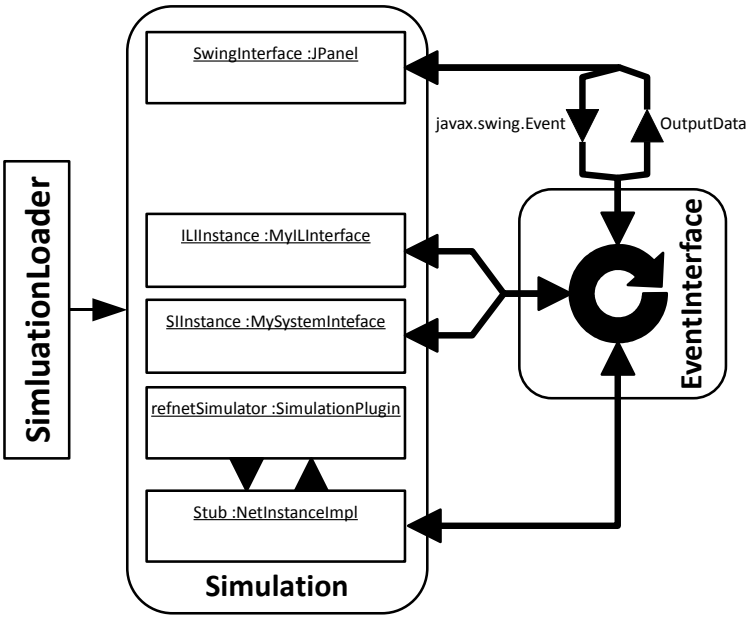


Figure 5.12.: Components and data objects used for simulation and controlled by the *EventInterface*

identifying the correct interaction element for appropriate and accurate replacement (step ❸).

The UIEditor's reconfiguration module is based on the implementation of the PhysicalUIEditor, with the exception of the palette for adding interaction elements. Still, by double-clicking interaction elements, the user can change physical parameters of the interaction elements, including their position and size. Reconfiguration can be applied by using the buttons of a toolbar added at the top of the visual interface of the reconfiguration module. Figure 5.14 shows a screen-shot of the reconfiguration module. From left to right the following reconfiguration operations are implemented and can be used: (a) parallelization, (b) sequentialization, (c) discretization, (d) replacement and merging of interaction elements for output, (e) duplication, and (f) deletion. The button with the arrow icon can be used to restart the simulation using the newly reconfigured user interface. Before applying the reconfiguration operations to the user interface, however, the user has to select the interaction elements to be involved in the reconfiguration, for instance, the buttons labeled 'SV1' and 'SV2', by clicking on them and then pressing the parallelization button in the toolbar. A new button will be added to the user interface that triggers the parallelized interaction processes. The user may also have to enter other information, as is the case with sequentialization. Here, the interaction elements must be sorted into a list to define the sequence in which the associated interaction processes should take place. Here, too, a new button will be added to the user interface that automatically triggers the sequence of interaction processes.

The implementation of the reconfiguration module and the data files involved can be seen in Figure 5.15. The information resulting from the selection of interaction elements and the subsequent application of a reconfiguration operation, the *RuleGenerator* generates a DPO rule (cf. Section 4.2) involving the reference net in PNML format. This involvement is important because the *RuleGenerator* has to create the correct mapping to the net and probably has to duplicate parts of the original net (cf. Section 4.2). To extract specific interaction processes from the interaction logic as a whole, the *RuleGenerator* implements a simple algorithm for extracting

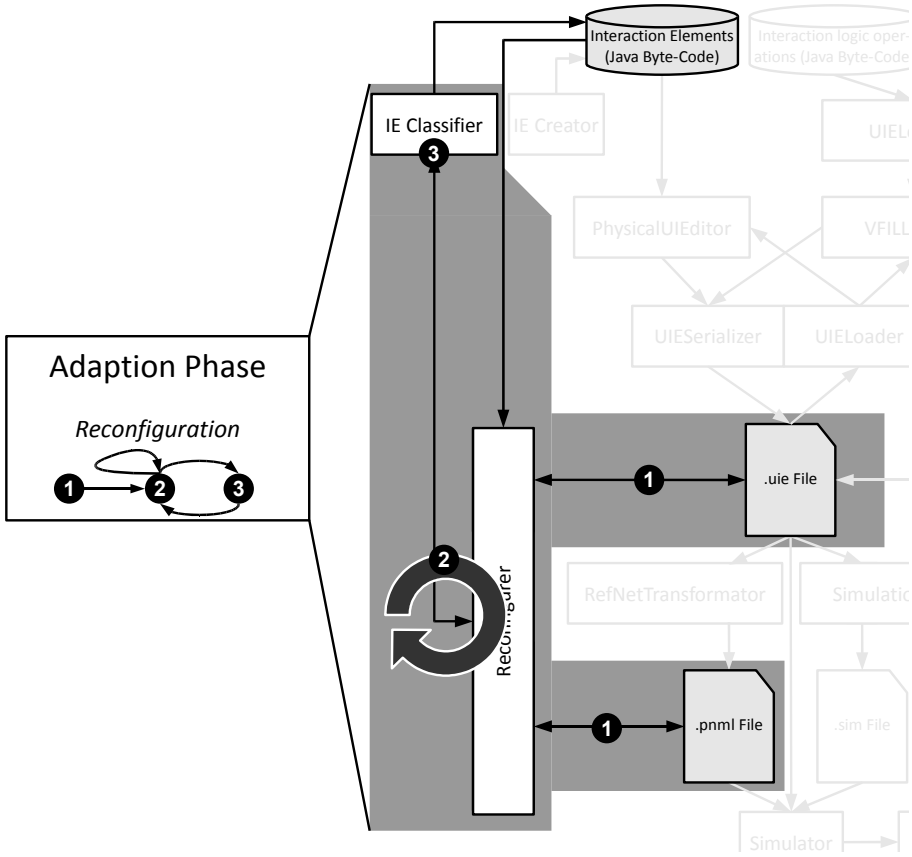


Figure 5.13.: Extraction of modules for reconfiguration of user interfaces using the UIEditor framework

the subnets of the specified reference net. Still, there is space to extend the algorithmic support for this kind of extraction of interaction processes. These aspects will be described in detail in the next subsection.

After a rule has been created, it is applied to the original PNML-formatted reference net using the implementation developed by Jan Stückrath [265], here subsumed by the *PNReconfigurer* component.

Interaction Process Tracing

Algorithm 5.1 (closely related to Java source code) traces interaction processes. The data structure required by the algorithm is a Stack, which is defined much as it is in the basic literature [101, 233], and a data structure representing a reference net, which has been read from a PNML file and is suitable as input to Algorithm 5.1. Figure 5.16 illustrates the data structure of a reference net and presents it as a UML class diagram. It is built on two basic types representing edges (class *Edge*) and nodes (class *Node*) in a reference net, where the type *Node* is further specialized to the types *Place* and *Transition*. Any *Node* or *Edge* has an inscription defining references to variables of a specific type. Instances of the classes *Place*, *Transition*, and *Edge* are accumulated in a object of type *ReferenceNet*.

Based on this data structure, the tracing algorithm can be implemented as in Algorithm 5.1. The *ReferenceNet* R serves as input for the algorithm and contains the interaction process to be traced, the starting transition node t, and the process *id procID* of the interaction process

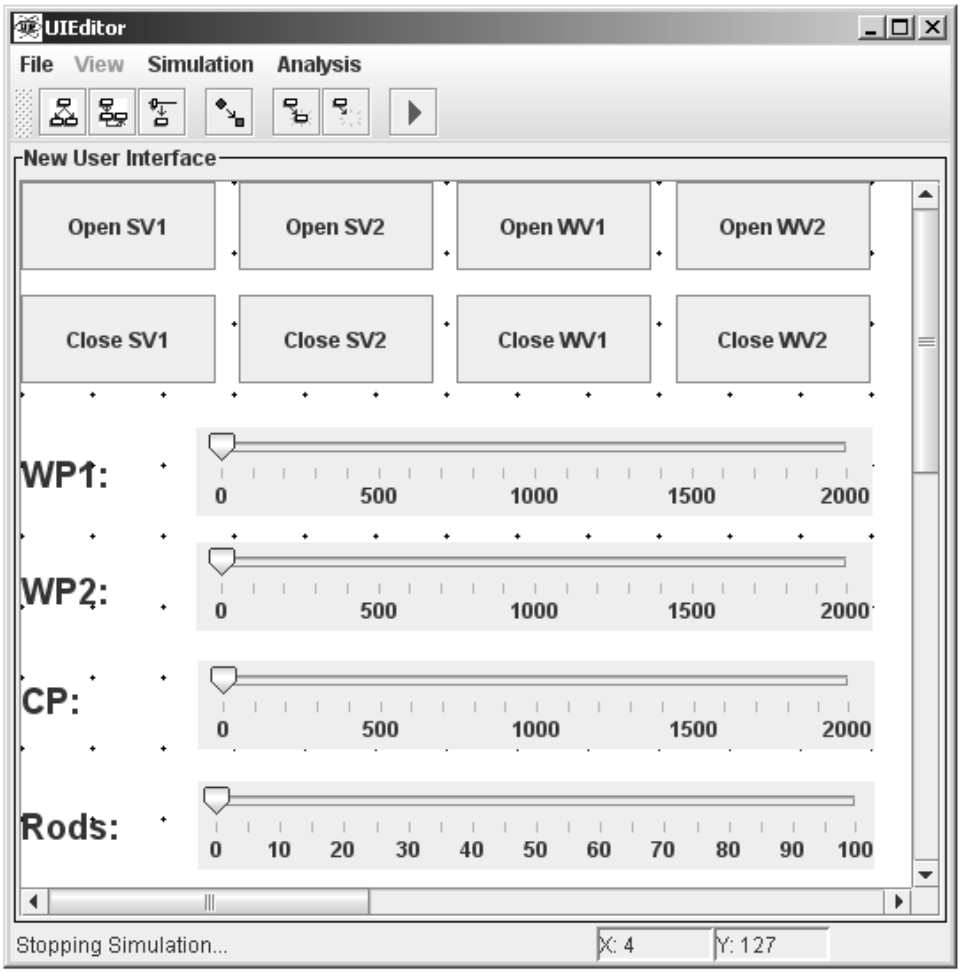


Figure 5.14.: Screenshot of the visual interactive interface for reconfiguring user interfaces

to be traced. The transition t can be derived from its associated *Widget* by the *WidgetEvent* object associated with that widget and derived from the `.uie` file. The process *id procID* can be derived from the outgoing edge of transition t. Resulting from the transformation algorithm in Section 3.4, any *WidgetEvent* is transformed into a transition node connected by an outgoing edge to a place node. Thus, this transition always has only one outgoing edge with an inscribed tuple, where the first element of the tuple defines the process *id* of the process that is triggered by this transition.

This algorithm generates a *ReferenceNet trace* that holds only those elements of R that are part of the interaction process, with process *id procID* triggered by transition t in reference net R. To generate *trace*, a *Stack edges* is instantiated that holds all outgoing edges of a current node in the algorithm that is part of the interaction process. Any edge in *edges* will be processed iteratively. If the edge has the same process *id* in its inscription or a variable for the process *id*, the target node is part of the process. The outgoing edges of the target node will be added to *edges* for further processing. The inner **IF** clause of Algorithm 5.1 handles the case if the node references another node with no outgoing edges. These nodes are, for instance, transitions representing an interaction logic operation or a system operation, where the process is redirected to a Java implementation and returns to the reference net. Interpreting such references also identifies channel operations.

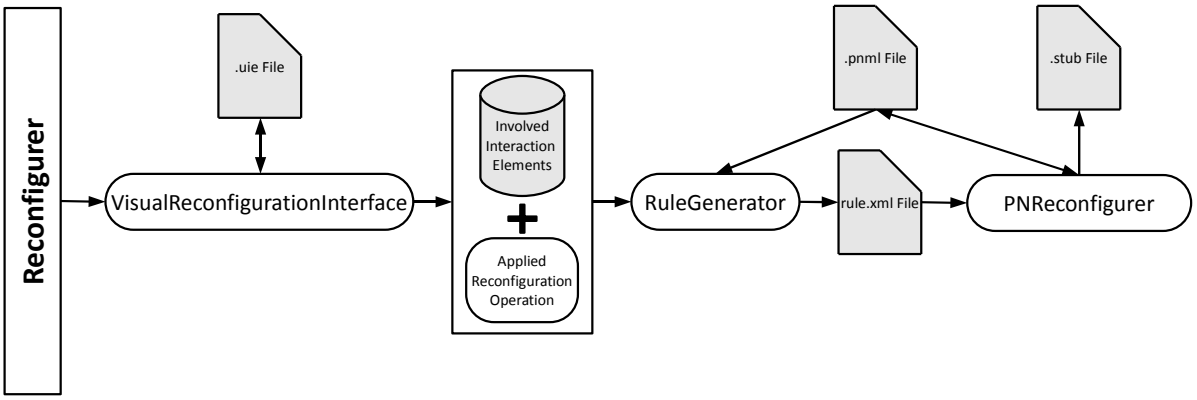


Figure 5.15.: Components and data objects that are used for simulation and are controlled by the EventInterface

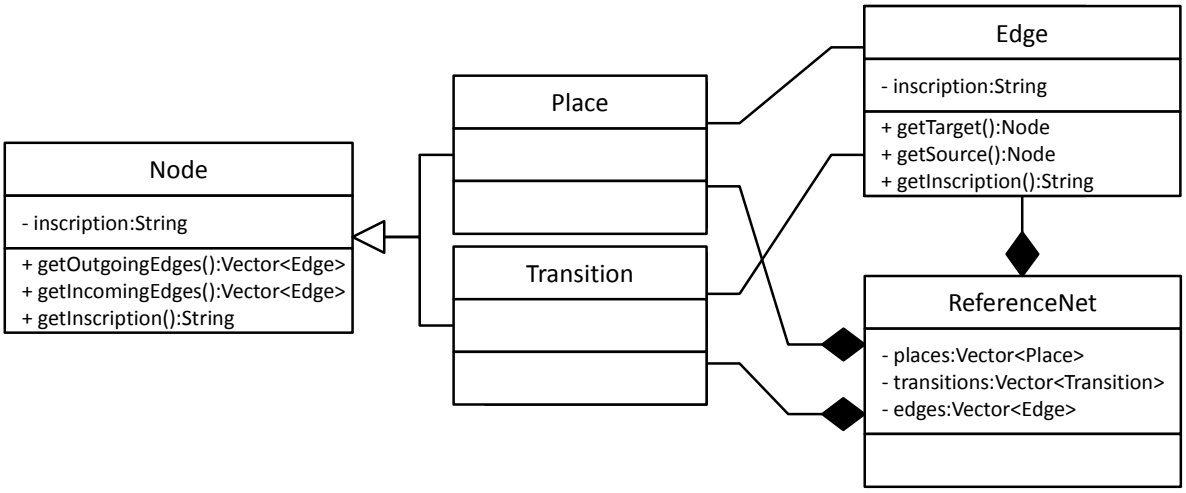


Figure 5.16.: UIEditor framework's data structure, representing a reference net for processing using an object-oriented programming language

For interaction processes that process data for output, the algorithm is almost the same. It only changes in that incoming rather than outgoing edges and the target rather than the source node are investigated.

Figure 5.17 provides an example of how Algorithm 5.1 is applied to an interaction logic. The interaction logic shown is the same as that shown in Figure 3.28 in Section 3.4, where the various functional parts of a transformation can be seen. In Figure 5.17, the result of applying Algorithm 5.1 to this partial interaction logic is identified by the black transitions and places. The first element in this trace results from the transition with the inscription `:m_input5974981199492591878(var3)`, which identifies an event of a certain interaction element in the physical representation. Thus, the partial interaction process traced in the diagram represents the interaction process that is triggered by the event identified by the id beginning with `597498119...`. Furthermore, it can be seen in Figure 5.17 that the process id is not stable throughout the interaction process. Thus, an interaction process is not identified by only one id but by a set of ids, where every set is pairwise disjunct to all other sets representing interaction

processes. The extension of the above algorithm to this fact is simple. The current valid ids identifying the interaction process have to be stored in an array or *Vector*, thus enabling the correct edges to be identified.

Algorithm 5.1 (Generating reconfiguration rules from inputted data)

(1) **INPUT** R, t, $procID$.

(2) Stack $edges$ = new Stack(t.getOutgoingEdges()).

(3) ReferenceNet $trace$ = new ReferenceNet().

(4) $trace$.addNode(t).

(5) **WHILE** $edges$ is **NOT** empty **DO**

Edge e = $edges$.pop()

Node $target$ = e.getTarget()

IF (e.getProcID.equals($procID$))

IF ($target$.getOutgoingEdges().size() == 0 & $target$.getReferencedNode() != null)

$edges$.push($target$.getReferencedNode.getOutgoingEdges())

ELSE $edges$.push($target$.getOutgoingEdges())

$trace$.addNode($target$)

$trace$.addEdge(e).

(6) **OUTPUT** $trace$.

Serialization

The main motivation for serializing a formally modeled user interface is to make it persistent based on a format that conforms with XML so that it can serve as a basis for the above-introduced simulation and reconfiguration workflows and processes. It was decided to develop a proprietary format based on XML to serialize a user interface that is modeled based on VFILL on the logical layer and combine it with a physical model without making compromises by using existing formats that may result in loss of information or overhead. Still, because this format, called *UIEML* (where UIE reference to the UIE in UIEditor and ML stands for Markup Language), is based on XML, it can be easily transformed or exported to other standardized formats, like UIML. Figure 5.18 shows the structure of the format. All file formats used are specified as RelaxNG schema in Appendix A as they were implemented in the UIEditor framework.

The root node `ui` has three basic child nodes describing (1) the physical representation (`design` node), (2) the interaction logic (`interactionlogic` node) as a FILL graph, and (3) Java classes (`programminterfaces` node) defining the connection to implemented interaction-logic operations (`ilopinterface` node) and to the system interface (`systeminterface` node). The physical representation of a user interface is further divided into interaction elements represented as a set of `widget` nodes, where every widget node is further split into its physical

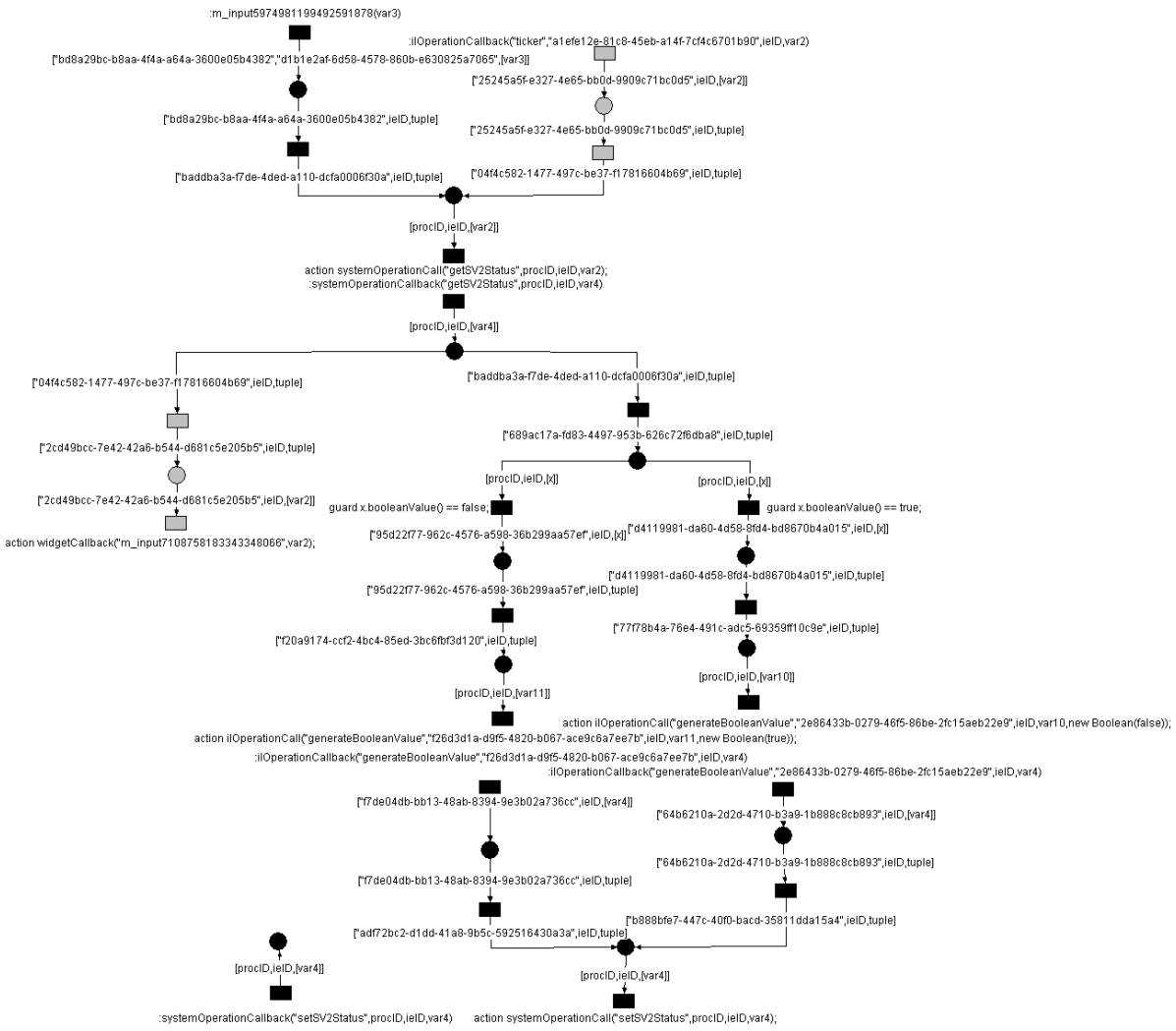


Figure 5.17.: Example of a traced sub-process of a given interaction logic as a reference net

parameters (`bounds` and `parameter` nodes) and an `interface` node. The interface node defines `widgetport` and `event` nodes as its children, which represent either values that can be set to the interaction element or events that are emitted by the interaction element. Furthermore, these two types of nodes define proxies in the interaction logic, where edges in the FILL serialization reference the *id* of these node types. The `function` nodes of the widgetport and event nodes define the Java method associated with the interaction element to be called on either the system or the interaction-logic implementation.

The `interactionlogic` node that represents the interaction logic of a user interface is subdivided into a set of `ilmodule` nodes, each of which represents one interaction process that is associated with exactly one interaction element. Thus, the set of all `ilmodule` nodes defines the entire interaction logic. Any `ilmodule` node is further divided into a set of operations (`operations` node) and a set of edges, called connectors (`connectors` node). The `operations` node has children of type `bpmn` representing BPMN nodes, `channeloperation` representing channel operations, `iloperation` representing interaction-logic operation, and `systemoperation` representing system operations in a FILL graph. All these nodes except, the `bpmn` node, have children, which represent (1) visualization aspects for a visual presentation

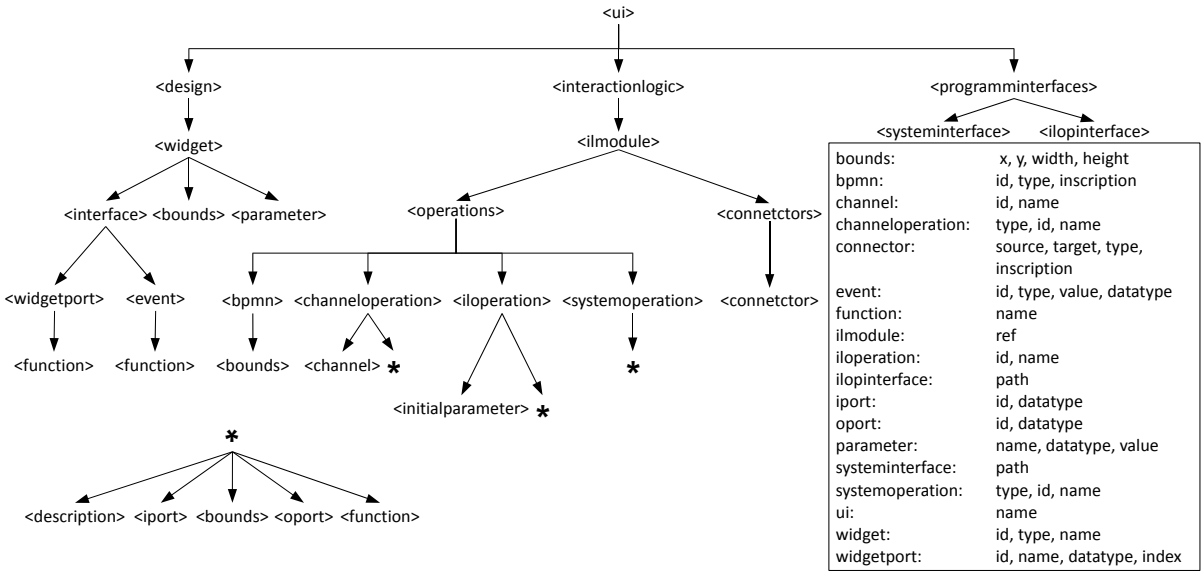


Figure 5.18.: Proprietary file format for serialization of user interfaces created using the UIEditor framework called *UIEML* (`.uie`)

as a FILL graph (`bounds` node), (2) the Java method they are associated with (`function` node), and (3) a description (`description` node). The input and output ports of FILL operations are represented as `iport` and `oport` nodes.

Interaction-logic operations also have child nodes of type `initialparameter`. These nodes define individual information attached to exactly this operation. For instance, an interaction-logic operation generating a fixed Integer value could be used several times in an interaction-logic model but should not always generate the same value. Therefore, the user sets a specific value for the interaction-logic operation stored in the `initialparameter` node. Furthermore, channel operations define a child node of type `channel` that specifies the channel they are connected to. BPMN nodes are only described by their position in a visualization as a `bounds` node. The type of any given BPMN node is defined by the `type` attribute of the XML node.

The box on the ride side of Figure 5.18 contains the attributes associated with these nodes. The values of `id` attributes are always unique for every node in a UIEML file. Other attributes use the `id`s of other nodes to reference those nodes. Examples are the `ref` attribute of the node `ilmodule` and the `source` and `target` attributes of the node `connector`. Most attributes are self-explanatory because they are used in the same way as in other XML-based description languages.

As mentioned above, the UIEditor framework implements components for reading and writing serialized data. Still, this XML-based representation uses an object-oriented data structure implemented in Java, as can be seen in Figure 5.5. In addition to being able to transform UIEML using, for instance, XSLT, this representation can also be used to generate and serialize other formats. One example of a possible serialization of a user interface or, more specifically, its physical representation, is SwiXML as introduced and described by Paulus [213]. This format is an XML-based description language for Java's Swing API. In the first version of the UIEditor framework, it was planned to use SwiXML to serialize the physical representation. However, this plan was rejected because of the specific requirements to define input and output proxies.

5.3. Future Extensions

The UIEditor framework supports a broad set of modules for modeling interactively and visually formal user interfaces, storing them using implemented serialization components and XML-based description languages, creating and executing simulations, and applying reconfiguration on the basis of formal interaction logic and a visual and interactive editor. Three main areas still require further investigation and implementation: (a) the module for analysis, (b) the module for modeling interaction elements, and (c) a module implementing the use of distributed user interfaces for multi-person, multi-interface scenarios.

Analysis

It has been already said that analysis is a big issue in formal user interface modeling but still has not been investigated and implemented in a satisfying way. To this end, the following list offers some ideas concerning future implementation and the necessary inclusion of third-party libraries.

Verification and Validation One main issue is the verification and validation of formally modeled interaction logic. Possible approaches include the use of state-space analysis from process investigation in higher Petri net studies. Furthermore, analysis of reachability in state-space analysis could be another possible extension.

Log and interaction analysis Some work has already been done in the analysis of log data. Still, future work might include interaction analysis during runtime. Paired with learning algorithms and powerful knowledge representation concepts, these analysis methods could be used, for instance, to identify errors during runtime and generate reconfiguration suggestions.

Formal usability A further field of investigation and implementation is that of formal usability. The next step here would be to investigate the degree to which usability can be defined using a certain formalism, like Petri nets, and how this kind of usability might be checked against a (formal) user interface model.

Online analysis and error detection Using data analysis techniques and formal modeling approaches, errors during user interaction with a system can be identified and used for triggering reconfiguration. This serves as a knowledge base for generating reconfiguration rules. Further insight into this subject will be offered in Chapter 8.

In general, an Analysis module as introduced in Figure 5.2, can be used in various processes and subprocesses. The results of this analysis can be used in a creation phase to support the modeler through such means as formal usability concepts. Furthermore, analysis can also trigger reconfiguration and serve as an information source for generating reconfiguration rules. In simulation, online analysis can also provide the user with additional information, for instance, by predicting situations that may occur during subsequent interaction.

Creation of Interaction Elements

In cooperation with Emeljanov [81], the author developed a formal approach to modeling interaction elements using a layered structure paired with reference nets, as described in Section

3.6, as an extension of the creation phase in the general working process. Still, tool support for interactive and visual modeling has not been fully developed. Emeljanov started to implement a visual editor that supports a click-and-drag-based canvas implementation for modeling the image and event boxes paired with a visual editor for modeling reference nets implemented in the Renew framework. However, further implementation still needs to be done to complete this basic implementation for the UIEditor framework. Besides finishing the visual editor, a data-structure accompanied by a serialization format still has to be developed to make the interaction elements persistent and usable for modeling, simulation, and reconfiguration.

Distributed User Interface Creation and Simulation

Up to now, the UIEditor framework implements modules providing tools for one-person, one-interface, one-system scenarios only, which is the case in most (historical) HCI research. One important issue in future work will be the transformation of formal concepts to multi-user, multimodal scenarios also supporting various instances of user interfaces as extensions of the working process introduced in Figure 5.3. For this purpose, the framework should offer extended modeling support, as well as extended simulation support for the distributive simulation of user interfaces. Modules for distributed simulation and distributed reconfiguration should implement various concepts known from research on distributed systems (cf. Tanenbaum et al. [266] and Coulouris et al. [49]), like distributed communication protocols (cf. Newman [185]) or transactions in distributed reconfigurations (cf. Moss [174]).

In this context of extension, the general workflow described above, which mainly addresses the one-person, one-user interface, one-machine scenario, further extensions to this workflow are also thinkable. Thus, questions of distributed interaction could extend the workflow, as could further implementations of mobile interaction. Furthermore, introducing the context for interaction, such as the surrounding environment (which up to now completely has been excluded from the scope of this dissertation), can extend the workflow in all three phases. Involving the environment explicitly in the modeling of user interfaces results in the need for further modeling approaches and tools. This is also the case with context-sensitive reconfiguration or simulation. Thus, a lot of future work remains to be done. Further aspects of future work are described in Chapter 9. All these extensions have to be supported by an implementation of multi-view user interfaces on modeling, simulation, and reconfiguration layer. This is still part of future work but has been considered in the basic software structure of the framework.

5.4. Conclusion

Every formal modeling approach needs effective tool support, as various examples show. For instance, UML is supported by a broad set of tools for visual modeling and translation to source code; one example is Poseidon[5]. Jeckle [123] provides a list of tools for UML modeling that shows how important tool support is for modeling languages. The same is true for Petri nets as the tool list[6] on the website *Petri Nets World*[7] shows. This chapter has introduced an implementation of a framework for the visual modeling of user interfaces using VFILL, as well as an editor for the visual modeling of the physical representation. Furthermore, various modules

[5] http://www.gentleware.com/
[6] http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html
[7] http://www.informatik.uni-hamburg.de/TGI/PetriNets/index.html

have been described for simulating and reconfiguring formally modeled user interfaces. Nevertheless, described above, many issues remain to be dealt with in future work. Here, modules for the analysis of formal interaction logic were briefly discussed, as were implementations for the creation of interaction elements and a module for modeling, simulating and reconfiguring distributed user interfaces based on the formal approach described above.

Chapter 2 through Chapter 5 described related work on which basis a formal approach to modeling user interfaces in a three-layered structure was defined, followed by an introduction to a formal reconfiguration approach for implementing adaptive user interfaces on a formal basis. This concept of the formal modeling and reconfiguration of user interfaces is completed by the implementation of the UIEditor framework, which was described in this chapter. The next step will be to evaluate this approach with regard to its application in the field. To do this, two major scenarios were identified and investigated. These investigations will be described in the next two chapters. As will be seen, this research showed that, as hypothesized human error in interaction can be reduced through user interface reconfiguration. The next sections will introduce the three relevant studies, which were conducted in cooperation with Prof. Dr. N. Baloian from the Universidad de Chile (study presented in Chapter 6) and with Prof. Dr. A. Kluge and Dr. D. Burkolter from the University of Duisburg-Essen, who addressed the psychological background for the studies presented in Chapter 7. These studies were also covered in lectures at the University of Duisburg-Essen.

6. User Interface Reconfiguration in Interactive Learning Systems

The investigation and development of interactive learning systems is a specific application area of human-computer systems. User interfaces play a central role in this area because they have to transmit the learning content to the learner. It has been shown that presentation and interaction with learning content directly influence learning success. As an extension of a basic one-learner-one-system approach, computer-supported cooperative learning systems integrate group learning aspects, thus enhancing learning success.

To describe the reconfiguration of user interfaces as an extension of learning processes using interactive learning systems, this section first provides a survey of related work in its thematic context (Section 6.1). In a 2006 student project [142], a learning system called COBO was implemented to support the cooperative learning of cryptographic protocols; this system has since been extended by connecting it to the UIEditor framework, resulting in an implementation called XCoBo (Section 6.3). To evaluate the impact of reconfiguration in cooperative learning scenarios, an evaluation study was conducted in 2009, which yielded positive results (Section 6.4). The main concepts of this chapter were published in [289].

6.1. Introduction

This dissertation was motivated by several questions. First and foremost, to what extent can the mental models of human users be transferred directly or indirectly to a user interface and, especially, to its interaction logic? Furthermore, to what extent does the adaption of a given interaction logic through reconfiguration to the user's mental model influence the success of the human-computer interaction, especially in the control of complex systems and processes. It has been shown in related research, that building mental models is, on the one hand, influenced by the experiences a person has had in the past and, on the other, by the study, training, and practice that person has done. Experience is gained primarily through living and working; thus, it is not normally directly controlled from outside. In contrast, study, training, and practice can be defined and developed didactically, as with children at school and students at university. Thus, there are procedural methods for building mental models through these didactic means. Regarding the formal modeling of interaction logic and the relationship between interaction logic and a person's mental model, a question arises: To what extent can building a formal interaction logic through reconfiguration construct and extend—or, put more simply, influence—a mental model such that both the interaction logic and the mental model at some point converge, becoming in the main equivalent? Thus, two hypotheses were formed: (a) an interaction logic built by a user through the reconfiguration of a given user interface is a representation of the mental model of the user and (b) this influences the success of any interaction between that user and that system.

To investigate these hypotheses, an evaluation study was conducted in 2009 at the University

of Duisburg-Essen, investigating the influence of user interface reconfiguration on the learning process of a given cryptographic protocol and on the construction of a mental model. Since the structure of most cryptographic protocols involves more than one participant, the investigation was embedded in a cooperative learning scenario. Thus, if it was possible to show that reconfiguring interaction logic has a significant influence on the learning of certain content, this would imply that interaction logic is a formal representation of the person's mental model concerning that content. In the study, students created a cryptographic protocol through reconfiguring a user interface, such that the resulting interaction logic implemented all necessary operations and data processing and the students were able to simulate the whole protocol. The user interface the students created was an image of their mental model of the protocol. If the mental model was erroneous, the interaction logic was also erroneous, and the students had to fix the problem through reconfiguration. Thus, learning success was implicitly a validation of hypothesis (a) because if there was no success, the mental model might be correct or incorrect, but its accuracy was unverifiable because the interaction logic was not correct. If the interaction logic was correct and the students were able to validate it, because the students had built the interaction logic from scratch, it verifiably represented the relevant parts of the students' mental models.

Hypothesis (b), which deals with the influence of reconfiguration on interaction, was investigated in a subsequent evaluation study described in Chapter 7, which covers errors in interaction.

6.2. Computer-Supported Cooperative Learning

The main goal of this section is to investigate the possible influence of reconfiguration on a mental model. Mental models are generated through learning processes that can be supported by computers (computer-supported learning, CSL). CSL is a broad research field, which also contains the area of cooperative or collaborative learning systems (computer-supported cooperative learning, CSCL) [100, 235, 288]. CSCL is of particular interest in the integration of user interface reconfiguration into learning systems, which results from further research on reconfigurable interfaces, or Concept Keyboards (CKs), which were evaluated in single-learner systems and scenarios [143, 144, 248]. The next step was a multi-user scenario, investigated and implemented by Kotthäuser et al. [142]. The combination of this multi-user approach with the UIEditor framework offered complex reconfiguration operations to learners.

A learning strategy for which computer support is especially applicable is problem-based learning (PBL) [141]. Here, the idea is to impart knowledge by having the learner apply various methodologies with varying degrees of computer support in order to solve specific problems. Problems to be solved can be of a theoretical or practical nature, depending on the learning content. In a collaborative scenario, theoretical problems can be modeled and discussed in the group using collaborative tool support. Often, theoretical problems are learning content in computer studies. For instance, cryptographic algorithms and protocols can be learned using a PBL-based approach. For instance, investigating security aspects of cryptographic protocols could provide the subject of a collaborative learning scenario. These aspects could be formally modeled in a first step by the learners and then validated or verified by the computer to generate feedback to the learners [161].

In application-dependent problems, learners are asked to analyze concrete issues in a cooperative manner. Here, the main goal of this analysis should be a solution that is not necessarily reached. The computer supports only tools for displaying the practical problem, changing its

representation, interacting in a certain way with it, solving identified sub-problems, combining various parts of a possible solution, and checking the correctness of a derived solution through automated verification or through interactive simulation.

The current literature shows PBL as applied to a practical or a theoretical problem. The idea of the approach described by Weyers et al. [289] is to combine these two different aspects in one consistent environment to learn complex algorithms. The learner first models the algorithm theoretically and then simulates it using a visualization tool to validate his model. In this way, the learner can assume two perspectives on one problem, combining both approaches in PBL. The first step in modeling, say, an algorithm, demands that the learner generate an overview of the entire algorithm and its functionality without being bound to a concrete use. In the second step, the learner is able to validate the mental model of the algorithm created in the first step by such means as visual simulation and comparing the outcomes to an expert model. This combination offers an iterative learning process that can be implemented using reconfiguration techniques for user interfaces. The generated interaction logic can be used for automatic verification of the learner's results and offer further information for his validation process.

As mentioned above, such a system was implemented using a system for visualization and simulation of cryptographic algorithms called CoBo for the practical part of PBL combined with the UIEditor framework, which supports the platform for reconfiguring the user interface to model the algorithm on a formal level. Figure 6.1 provides a model for how the reconfiguration of user interfaces in a CSCL environment can be applied. This model combines application-driven modeling (through the reconfiguration of user interfaces) and a simulation layer for informal validation paired with a formal validation based on Petri nets.

To model a cryptographic algorithm based on this architecture, the user has an initial user interface where every button is associated with a certain operation in the algorithm. Through ongoing reconfiguration by combining buttons to more complex ones, the learner implicitly models the algorithm on the formal level as a reference net. Using the connection to CoBo, the learner is able to validate the modeled algorithm by starting a simulation of the algorithm using the modeled user interface. If the algorithm terminates with the correct result, the algorithm was modeled correctly.

The idea of connecting interaction elements with conceptual elements in a subject to be learned has been investigated in other works on CKs. CKs are physcial or virtual keyboards that are easily adapted to different contexts. The primary feature of a CK is that keys trigger assigned concepts in given contexts. CKs have already been successfully used in various implementations (excluding reconfiguration) and evaluations of learning systems for developing single-user scenarios supporting the learning of complex algorithms [15, 17].

To benefit from the positive impact of cooperative work in learning [259], informal validation on the application layer has been implemented as a distributed simulation. To this end, CoBo implements a module for the distributed simulation of cryptographic algorithms, such that every involved party in, say, a key exchange protocol is controlled by one learner using CoBo in a cooperative way. Through reconfiguration based on the UIEditor framework, a touch screen can also be used for collaborative modeling of the algorithm.

In brief, the XCoBo system was implemented as a tool for a cooperative learning process that will be described below in greater detail. This learning process is based on the combination of two conceptual approaches familiar from PBL: the theoretical and the practical problem-solving approach. The entire learning process combined with the described tools was evaluated in an

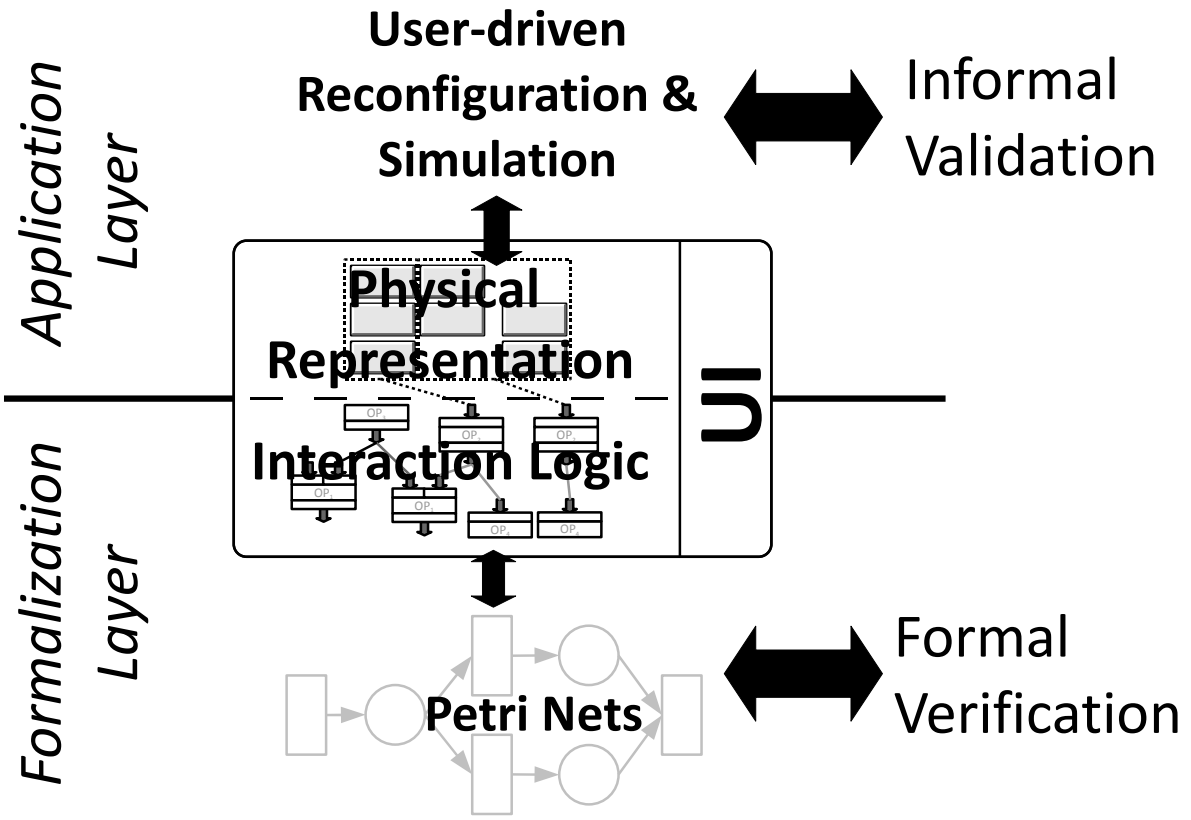


Figure 6.1.: Application of reconfiguration of user interfaces in computer-supported cooperative-learning

2009 study involving 66 students at the University of Duisburg-Essen.

The main motivation for extending Kraft's single scenario approach for learning complex algorithms [144] is the positive impact of group learning on success in learning complex algorithms, as has been shown in work by Nickerson [187], Webb [277, 278], and others. Working in small groups offers the opportunity to share insights (Bos [30]) and observe others' problem-solving strategies (Azmitia [11]).

The primary research focus in investigating environments for cooperative learning consists of dealing with modeling tools that support cooperation, cooperative design, knowledge building, and such issues as awareness (see Mühlenbrock et al. [176], among others). For the development of a collaborative learning tool for learning cryptographic algorithms, works dealing with algorithm visualization and simulation are of particular relevance. Many systems for algorithm visualization have been implemented, as described by Diehl [59, 60], and Stasko [263]. The positive impact of visualization on the understanding of complex systems and machines as a more general perspective on an algorithmic problem is described by Diehl and Kunze [61]. In addition to systems for algorithm visualization, several repositories for algorithm visualization have been developed (e.g., Crescenzi et al. [52]). The work of Shaffer et al. [249] gives an overview of state of the art for algorithm visualization. Solutions for algorithm visualization paired with concepts for explanation are described by Kerren et al. [133] and others. Shakshuki et al. [250] introduced an explanation system based on hypermedia concepts for web-based structures. Eisenberg [78] presented an example of basic research in algorithm visualization. In addition to the visualization of algorithms, their animation is an important aid for clearly

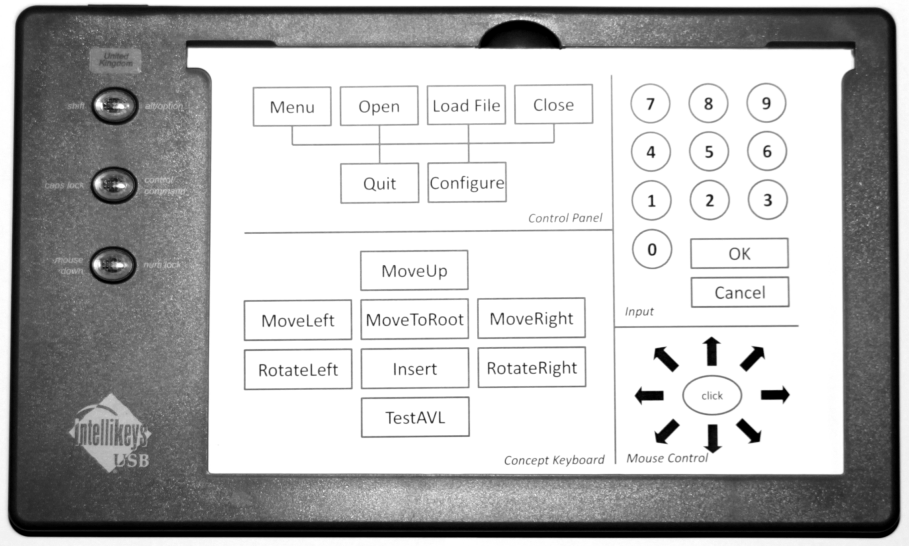


Figure 6.2.: Example of a concept keyboard used to interact with an animation of the AVL algorithm

demonstrating how the algorithm and its several parts work together. Brown [34] introduces and analyzes a taxonomy of systems for generating algorithm animation. Stasko [261] introduces a framework called TANGO for algorithm animation, which is extended [262]. A further overview of program visualization and algorithm animation is provided by Urquiza-Fuentes and Velazquez-Iturbide [271].

The notion of using the reconfiguration of user interfaces based on a visual and interactive editor was motivated by the use of CKs in the context of computer-supported learning. CKs have proven to be helpful tools in teaching complex algorithms, as shown by Baloian et al. [14, 18]. Another application area is the use of CKs in learning systems for blind and partially sighted children. Douglas and McLinden [68] describe the use of CKs for teaching tactile reading to blind children. This example shows the high flexibility of CKs. A physical implementation of a CK can be seen in Figure 6.2. This CK is equipped with a touch-sensitive surface, which can be covered by a simple piece of paper showing the concept keys. As the example in Figure 6.2 shows, every key is connected to a concept, which is itself associated with a concept in the animated algorithm. The use of paper makes the appliance of CKs highly flexible. A CK can be easily transformed into a software-based implementation using the UIEditor framework. The design can be changed by the visual reconfiguration module, which also offers reconfiguration. By combining the idea of CKs and formal reconfiguration in this way, the concepts associated with the concept keys can be adapted by the learner to his individual understanding and learning conception.

Reconfiguration as a concept in learning systems has also been investigated by Hundhausen [116]. He concludes that software for algorithm visualization improves the level of comprehension and, more importantly, that "what learners do, not what they see, may have the greatest impact on learning." These findings support the notion of combining algorithm visualization and animation with the use of CKs extended through the reconfiguration of user interfaces, which provides an experimentation aspect to algorithm learning. Colella [46] showed the impact of collaborative modeling in learning applications, which offers further support for the use of reconfiguration techniques as a tool in collaborative learning environments.

The implementation presented in the upcoming section focuses only on the input step in controlling a cryptographic algorithm. This should be extended to reconfigure the output side of the user interface—here the algorithm visualization or animation—as well. It has been shown that multimodal user interfaces have a positive impact on learning success in learning complex algorithms, as shown in [16, 17]. Thus, an important subject for future investigation is the reconfiguration and adaption of algorithm visualizations and animations; initial work in this area is described by Schmitz [242].

6.3. Simulation Environment for Cryptographic Algorithms

The use of user interface reconfiguration in learning systems will be described by introducing the CoBo system and then explaining its connection to the UIEditor framework, which resulted in the implementation of the software tool called XCoBo.

The Learning Process

Before explaining how reconfiguration techniques for user interfaces are used as extensions of CoBo, it is first necessary to explore their use in learning systems in general. Here, the idea is to create the CK as the first step in a learning process, followed by a validation step in the form of a simulation. If an error occurs during the simulation, the CK has to be reconfigured in such a way that an error-free simulation of the algorithm becomes possible. Figure 6.3 describes the workflow of a small group of learners. After being introduced to the protocol (which has to be learned) and the learning software, the learners are separated into small groups that reflect the number of participants in the protocol. They start by creating the CKs for each role in the protocol cooperatively, adding operations to keys on a (software) CK. Next, they define the role in the protocol that each key is associated with. Thus, the learner who represents a certain role sees only those keys that are associated with his role during simulation. After they have finished creating the CK, they start to simulate the protocol. If errors occur during simulation—for instance, if a certain operation is missing—they go back to the creation editor and change the initial keyboard by adding, deleting, or reconnecting the CK keys to other operations and roles. Once the simulation has been successfully completed without errors, the learning session is closed with a concluding evaluation.

CoBo

CoBo is a learning software that was developed in a student project in 2006 [142]. It was implemented to offer a tool that provides distributed simulation of cryptographic protocols, including algorithm animation and CKs, to control the simulation by the learners. Here, each learner is associated with a specific role in a cryptographic algorithm and can execute the operations associated with this role using the keys on the CK. For instance, if Alice should send a message to Bob in a certain situation in the protocol, the learner who represents Alice has to press the key to send the message to Bob and so forth.

The main application scenario for cryptographic algorithms and protocols is the secure exchange of information over insecure communication channels or to solve problems that are closely related to the exchange of information. For instance, most encryption algorithms need keys that have to be shared before the communication starts. Therefore, key exchange protocols were developed involving the two communication partners (often called Alice and Bob) and

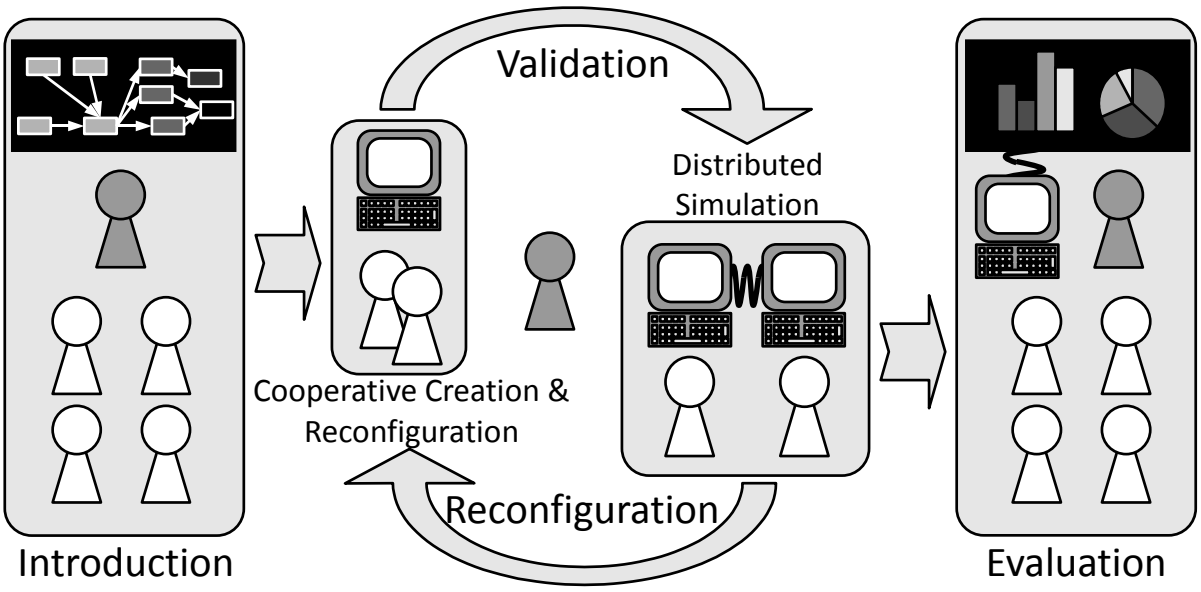


Figure 6.3.: Workflow for using CoBo for interactive visualization of cryptographic algorithms paired with a tool for user interface creation and reconfiguration to create CKs

in some cases a third party that shares a secret with both communication partners called the *trusted third party* (TTP). Especially in the context of learning such protocols, a distributed learning environment like CoBo is helpful. CoBo offers a framework for simulating and visualizing cryptographic protocols. Furthermore, it provides interfaces that allow the simple implementation of new protocols as Petri nets that describe the protocol formally. Furthermore, CoBo implements error handling. It offers a basic implementation structure for the rich animation of cryptographic protocols involving up to four communication partners. CoBo has also been transfered to mobile devices, as described in [18].

In order to use CoBo, the learner has to connect his CoBo instance to an existing learning session that defines the protocol to be learned. This session has to be started in advance by using another CoBo instance. The session runs on an instance of the CoBo Server that handles the distributed communication between all CoBo instances. The learner first connects to an existing session and then selects a role; after all participants have been connected to a session with their CoBo instance, the simulation begins. When the CoBo user interface (shown in Figure 6.4) appears, it contains only the operation keys of the particular role the learner selected beforehand and the initial visualization of the protocol. Next, the simulation starts, and the learner who is associated with the active role in the protocol (indicated as a pale icon in the algorithm visualization) can execute his operation keys. The animation associated with this executed operation is shown to all the learners. If the learner with the active role tries to execute an operation that is not correct in a given situation or state of the protocol, an error message appears informing him of his mistake.

When considering cryptographic protocols instead of exploring an algorithm, as implemented in a learning system called ConKAV [17], more complex supervision is needed during simulation and interaction with the system. This is even more critical in a distributed scenario. Complex system models or action logic is necessary to meet the requirement of the complex behavior of the system and error recognition in the user's actions. This led to the use of Petri nets to implement structures for handling incorrect inputs by the learner. These modeled error

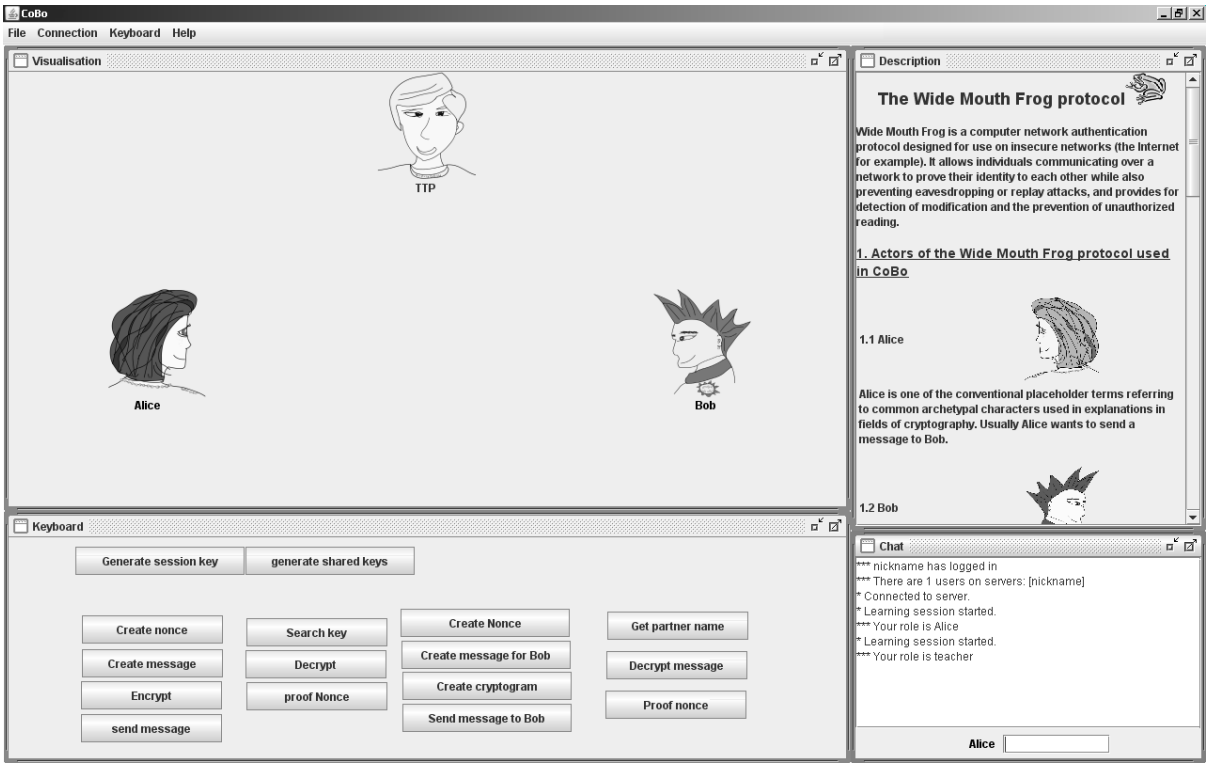


Figure 6.4.: CoBo's user interface during simulation in Needham and Shroeder's key exchange protocol

cases were connected to messages that were presented to learners who tried to execute incorrect operations in a certain situation of the protocol. To this end, CoBo implements a simple simulation environment for Petri nets with a referencing mechanism to Java code like that of reference nets but on a much simpler level. It extends a basic Petri net formalism closely related to S/T nets by *error places*, which are connected to special nodes called *message boxes*. Depending on the number of tokens contained by an error place, an error message is sent to CoBo's user interface. In this way, it is possible to model a scaled message system reacting to any incorrect execution of operations. Transitions can be called from Java code by sending a fire event to the net indicating the label of the transition that should be fired. If it is enabled, it fires. If it is not enabled, an *else transition* can be defined that is used, for instance, to add a token to an associated error place.

Using Petri nets for modeling action logic combined with an error model enables more complex supervision during simulation and interaction. It could be also used to model action logic for systems like ConKAV that have been implemented for algorithm learning, for instance to learn special types of sorting algorithms (Bubble sort, etc.) [17]. This basic, proprietary developed Petri net format can easily be implemented using reference nets. During the development of CoBo, there was no necessity to use a more complex and powerful formalism than the proprietary one described above.

Besides the Petri net-based module for simulating action logic, which models the protocol, CoBo offers interfaces for algorithm visualization and animation. The major component of the algorithm visualization module is the canvas component that combines image-based animation embedded as GIF animation and algorithms for animation implemented in Java. These two

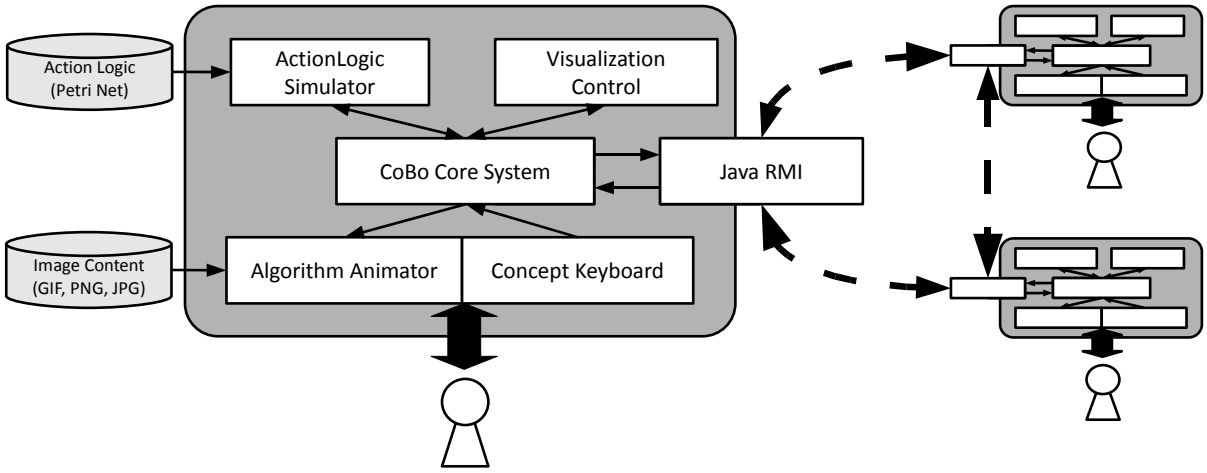


Figure 6.5.: CoBo's architecture, showing a standard example of communication between three CoBo instances in simulation mode

main tools can be combined in a complex algorithm animation. The main focus in CoBo was the animation of cryptographic protocols for distributed simulation. Three aspects of algorithm visualization were of special interest: (a) visualization of two or more participants in a protocol, (b) visualization of the current knowledge the participants in the protocol have in a certain situation (e.g., keys, messages, cryptograms), and (c) animation of the sending of messages between participants. Aspect (a) is supported by adding images to the algorithm description in XML that are automatically positioned and resized in the canvas. For aspect (b), knowledge items like keys and messages are visualized using pixel graphics that are positioned automatically under the images of the participants defined in (a). To animate the sending of messages between participants (aspect [c]), Java-based implementations move images in the canvas. Figure 6.4 describes the initial status of a visualization canvas: the Wide Mouth Frog protocol with three participants—Alice, Bob, and the TTP.

Figure 6.5 shows the entire CoBo architecture. It combines the formally modeled action logic as a simulation module that is directly connected to the implementation of the protocol. This implementation offers functionality for the communication between the visualization module and the CK. Interfaces for this kind of communication are offered by a core system, which also implements the entire communication infrastructure for distributed simulation of the protocols. The communication architecture is implemented on the basis of Java RMI[1]. Selvanadurajan [248] extends this basic implementation through the addition of a help desk system. It has been evaluated in a study of usability aspects involving computer science students at the University of Duisburg-Essen with positive feedback.

Based on this flexible architecture for implementing distributed simulation of cryptographic protocols paired with a potent algorithm visualization and animation framework, further implementations are easy to deploy to the system. Kovácová [143] implements a new protocol for CoBo in three steps, which are also suitable in all other cases: (a) modeling the action logic as a Petri net, (b) creating images for visualization and animation, and (c) implementing the code modeling the communication between action logic and visualization. Finally, these three elements in (each) implementation of a CoBo protocol are in one of the following formats: XML

[1] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

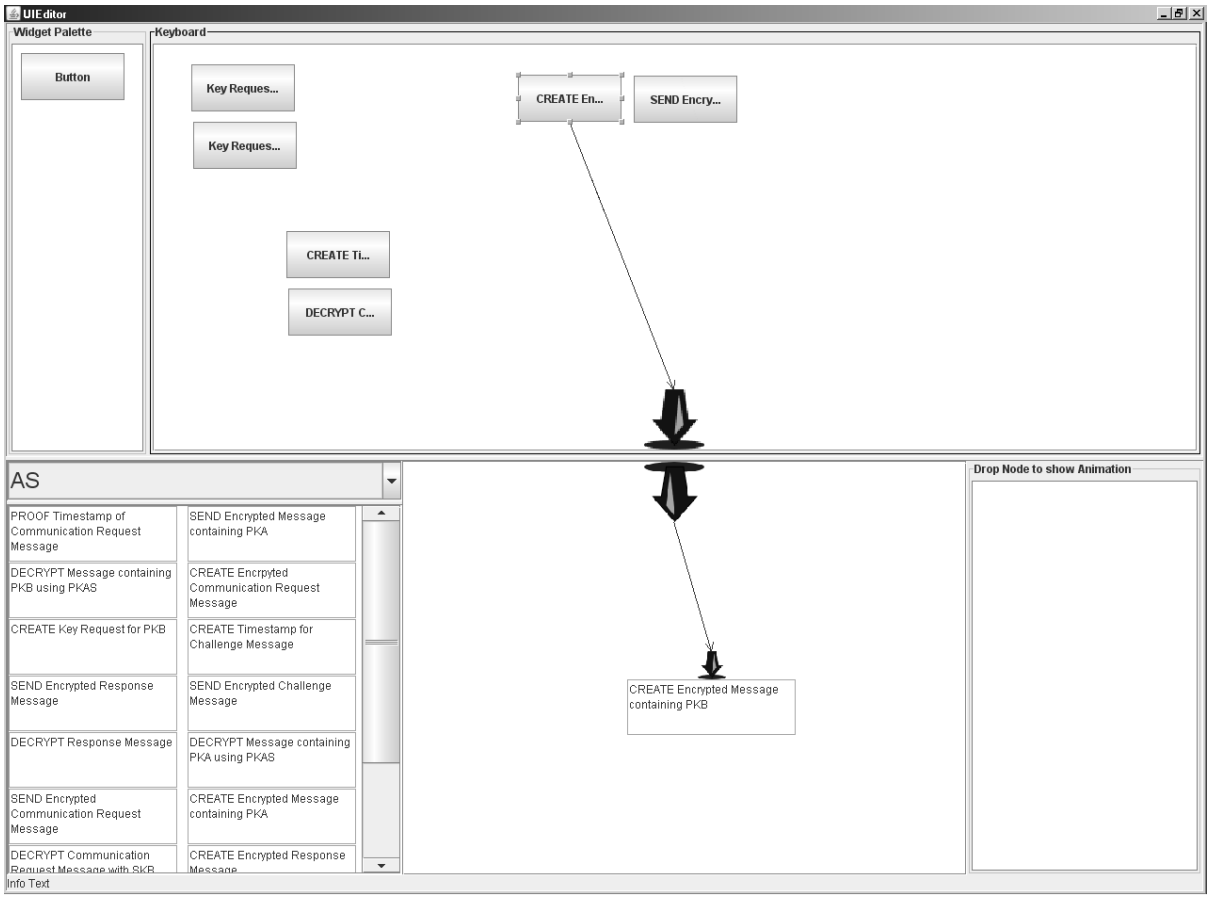


Figure 6.6.: Special implementation of the visual editor in the UIEditor framework for creating and reconfiguring CKs in CoBo

(action logic), Java code, and image data defined by a scenario definition in XML format.

Extension of CoBo by the UIEditor Framework

In order to create and reconfigure CKs in the above-described learning process (cf. Section 6.3), the UIEditor framework was added to CoBo. The resulting combination is called *XCoBo* and was used in the evaluation study that will be described below. The visual editor in XCoBo is based on the standard implementations of the UIEditor framework (cf. Section 5.2) and can be seen in Figure 6.6. It is divided into two main areas: (i) a canvas for creating the physical representation of the CK (seen in the upper portion of Figure 6.6), which also offers a palette for adding keys, and (ii) the interaction logic editor (seen in the lower portion of the figure), which is trivial here. Thus, any key's interaction logic consists only of a single system operation that triggers a certain operation in the protocol. Furthermore, the interaction logic editor includes an area that shows an animation of the operation, thus explaining what a given operation does in the protocol. The animation appears when the learner drops an operation node in this area. The operation will automatically relocate to its prior position. A second feature, which is different from the basic implementation of the UIEditor framework, is the combo box. This is where the desired role to be associated (or combined) with the selected key. In the figure, this combo box appears above the operation palette.

A further extension to the UIEditor framework was implemented to export the newly created

CK to a format similar to SwiXML [213], which can be easily read by CoBo. By loading a scenario that uses this modeled CK, it will appear in the lower part of CoBo's user interface automatically embedded in CoBo's simulation engine. In the case of an erroneous CK, learners return to the modeling editor as shown in Figure 6.6 and apply changes to it before restarting the simulation to validate the reconfigured CK.

6.4. Evaluation Study and Results

The following explanations, statements, descriptions, and scientific results appear here as published in Weyers et al. [289]. It was decided to reproduce the material in order to avoid unintended changes in the results of the 2009 evaluation study conducted at the University of Duisburg-Essen.

Since earlier studies with CoBo have already dealt with the use of CKs for supporting learning algorithms, in this section, the focus is on a formal evaluation of the above-described iterative and cooperative approach to the creation and reconfiguration of CKs in the context of cryptographic algorithms. Thus, the following aspects are important: (a) finding out how successful this learning method is in helping learners to achieve their learning goals, and (b) how motivating the learners find this approach and the impact of cooperation on the learning process. Another point to evaluate was the usability of the UIEditor framework in creating and reconfiguring CKs running on touchscreen interfaces.

Previous research has shown that, on the one hand, the use of CKs in algorithm simulation and visualization is both effective and motivating [14]. On the other hand, it has been shown that extensions of this approach that include cooperation [144] and distribution [142] also support learning success and motivation [248]. After this previous work, the next step was to compare the efficacy of the cooperative CK creation process as a learning tool with that of using static and previously developed CKs to simulate the algorithm. Thus, learning success and motivation remain of primary importance.

In the context of the formal creation, redesign and reconfiguration of user interfaces, besides learning considerations, the usability of the creation and reconfiguration software is of central interest. Without a usable tool that enables the iterative creation of CKs through the use of a cooperative interface, the whole approach would be unsuccessful, with negative consequences for learning success. Further points of interest are the usability of the interface for creating CKs, the acceptance of the workflow, and the pieces of software involved as entities.

Based on these considerations and the earlier work, four hypotheses were formulated to address the following: (1) learning success, (2) motivation, (3) usability and (4) acceptance of the new approach to learning cryptographic protocols.

> **Hypothesis 1:** Cooperative/autonomous creation of concept keyboards supports the understanding of cryptographic protocols and enables the solving of more complex questions.
>
> **Hypothesis 2:** The iterative creation process motivates the learning of cryptographic protocols.
>
> **Hypothesis 3:** Interaction with the software to create the keyboards is intuitive and simple to learn.
>
> **Hypothesis 4:** The iterative creation process is preferred to other approaches.

As part of the undergraduate studies of computer science at the University of Duisburg-Essen, teaching algorithms using visualization tools is a primary goal. Especially in the context of cryptographic algorithms, this is a significant issue. During the summer term of 2009, this approach to cooperative learning of cryptographic algorithms was implemented and used in an algorithm and data structure lecture to give students the chance to practice for their exams and get feedback for further improvement.

Evaluation Study Setting

The testing was done in 15 sessions; every session had one or two learning groups and each group comprised two to three students. There were two types of groups: cooperative (test group) and non-cooperative (control group). In total, there were 32 students in cooperative groups and 34 in non-cooperative groups. Each session was organized in the following way: First, the entire group of students received an introductory presentation on the asymmetric Needham-Schroeder protocol and the two different software components to be used in the testing (UIEditor's visual editor and CoBo for simulation). The presentation and the lecture on the learning content were written out in full and read out in the same way for each group and in every session in order to assure that all the students received the same information. After the introduction, the students were split into two groups to start the testing activity. Students in the cooperative group had to build the asymmetric Needham-Schroeder protocol by creating a CK for the CoBo simulation environment in a cooperative way. The students in the second group—the control, or non-cooperative group—had to create the CK working independently, without the option of communicating with one another or working together on a common interface. Each student was assigned one explicit role in the protocol. The students in the non-cooperative group only had to create the interface for their role-dependent part of the algorithm; the students in the cooperative group created the whole keyboard for every role cooperatively. All the students had 40 minutes to create the keyboards and simulate the protocol. If errors in the design of the CK occurred during the simulation, they had to go back to the creation phase. As additional aid, every student received a written description of the protocol as a handout. The written information they received combined with the information from the introduction was sufficient to complete the exercise.

After the 40 minutes, all the students had to solve a second exercise individually (whether they were members of the cooperative or the non-cooperative group), for which they had ten minutes. The exercise was formulated in a printed document. The document described first how an intruder can break into the communication and then presented seven possible amendments to the protocol to avoid this problem. Two answers were feasible solutions; the other five answers were not correct. After the second exercise, the students had to fill out a questionnaire designed to validate or reject the four hypotheses presented above.

The questionnaire was broken down into the following eight parts:

1. Demographic data (five items)
2. A self-assessment concerning pre-knowledge of key exchange protocols as well as past participation in a lecture on cryptography (two items)

   ⇒ **These items were filled out before the students started the testing.**
3. Post-test, textual description of the problem on an intruder and seven possible answers

4. General statements concerning the exercises and experience with the software (three items)
5. Statements concerning the usability and understandability of the UIEditor (six items)
6. Statements concerning the workflow and work settings (seven items)
7. Statements concerning general issues with the workflow and motivation (five items)
8. A final question in which the students had to sort five different styles of learning on a five-point scale from 'I like using it' to 'I don't like using it'.

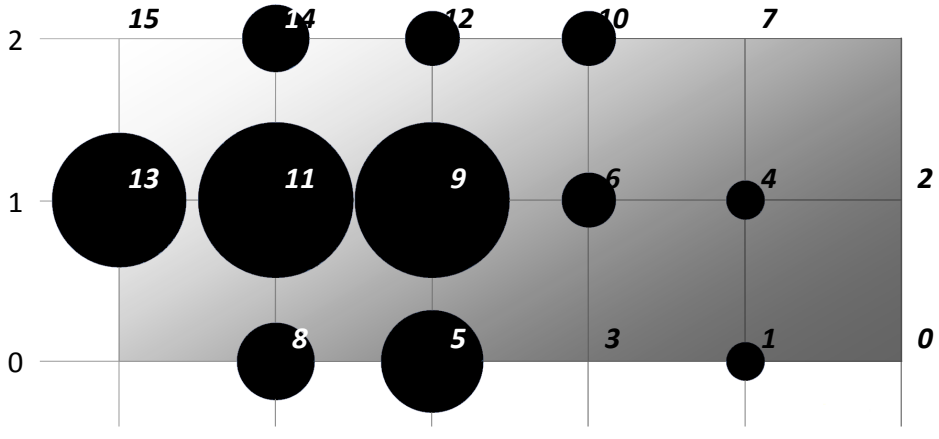
Concerning the statements in points 4 to 7, the students were asked to what extent these statements apply to them or not on a five-point Likert scale, ranging from 'fully applies (++)' through 'don't know (0)' to 'does not apply at all (–)'. The textual coding of the scale was adapted in the evaluation to a numbered scale from 5 (for total agreement) to 1 (for total disagreement). The neutral position was 3. The questionnaire was provided simultaneously in German and in English. For the evaluation of the hypotheses, there were different numbers of statements in points 4 to 7. For Hypothesis 1 there were 7 items; for Hypothesis 2, 1 item; for Hypothesis 3, 6 items; and for Hypothesis 4, 4 items.

A pre-test for measuring students' knowledge about the subject was not necessary because they were all in a second semester course (first year) and had never attended a lecture on cryptography, so their initial knowledge was almost nonexistent. This was confirmed by the answers they gave to the two related questions on the questionnaire: Only five people out of 66 said they had previously learned something related to this in a lecture. Also, regarding the self-assessment of prior knowledge about cryptography, the mean showed a value of 2.13 (Median 2), which indicates deficient prior knowledge (on a scale from 1 to 5, 1 for poor prior knowledge, 5 for very good prior knowledge).

Results and Analysis

In this section, the results of the testing and the questionnaires will be described and analyzed. The following abbreviations will be used: M stands for mean value, Med for Median, SD for standard deviation, t for t-value in a t-test and p for the probability of error. In some cases the interval of possible values for responses is given in square brackets [1, 5]. If no interval is given, the Likert scale [1, 5] is assumed [162]. In every case the values are integers and have a distance of 1. Other abbreviations will be introduced if necessary. All statistical analysis was conducted using the SPSS© tool in version 17 (PASW Statistics 17©, SPSS Inc.) [90]. The demographic data was analyzed using Microsoft Excel® spreadsheets [170]. Negative statements concerning the validation of a hypothesis were inverted for the statistical analysis so that high values always reflect a positive evaluation of the respective hypothesis. The responses to all statements concerning one hypothesis were summarized in one variable and separated only for the cooperative and the non-cooperative group. To evaluate the difference between the values assigned by the members of the two groups, a two-sample t-test was applied to the variables. If this test showed a significant difference between the cooperative and the non-cooperative group, the results were evaluated separately.

The described approach was tested with the 66 students split into two groups. If students knew each other, they were put into the cooperative group to avoid social problems influencing the work result. Otherwise they were split randomly. Most of the students were in their



Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Number	-	1	-	-	1	7	2	-	4	16	2	16	2	12	3	-

Figure 6.7.: Evaluation of post-test

2nd semester (Median) of computer science studies. Students of mathematics or physics and students in an international program for studies in engineering and computer science (ISE) at the University of Duisburg-Essen were also involved. The students were between 20 and 37 years old (Median 22); 15 female and 51 male students participated. Of the 66 participants, 61 had not participated in a lecture concerning cryptographic protocols and algorithms, as stated above.

The post-test on solving the problem of the intruder was given to the students as the second part of the questionnaire. A feasible way to rate the students' answers was to reward correct ones and penalize wrong ones. It was important that correct answers raise the rating of the item, and that wrong ones lower it. The rating was done by writing down all possible combinations of answers into a coordinate system. The number of correct answers was indicated by the ordinate; the number of wrong answers was indicated by the abscissa. The rating was then generated by starting at the best possible result of two correct answers and no wrong answer (a rating of 15) with a zig-zag-pattern down to the worst possible answer of five wrong answers and no correct answer (a rating of 0). The results of the rating process are shown in Figure 6.7. The bold numbers in the diagram show the rating, and the size of the circles represents the number of students included into the associated result (derived from the coordinates in the diagram) of the post-test. The table translates the students' results in whole numbers. It can be seen that on the scale of 0 to 15 more students completed this exercise with a higher rating (values from 9 to 14) than with a lower one (values from 1 and 8). It was decided in advance that if a student selected no answers at all (0 correct and 0 wrong answers) or all possible answers (2 correct and 5 wrong answers), that test would not be rated. However, this situation did not occur.

Based on the results of the self-assessment concerning prior knowledge of cryptographic algorithms (on a [1,5] scale), which showed a significantly ($t = 18.74$, $p < 0.1\%$) low result with $M = 2.13$ ($Med = 2$, $SD = 0.93$) the post-test provided good results. The ratings showed a highly significant ($t = 27.77$, $p < 0.1\%$) mean of $M = 9.78$ ($Med = 11$, $SD = 2.86$). Reformatting the scale [0,15] to a scale of [1,5] by a conversion factor of $16/5 = 3.2$ resulted in a mean of

M' = 3.06 on the intruder exercise. By comparing the mean of the self-assessment with that of the post-test, it can be seen that the new learning approach resulted in an enhancement of nearly 1 point. However, it should be noted that there was no significant difference between the cooperative and the non-cooperative group (independent two-sample t-test: t = 0.33, p = 74.3 %). The two groups performed equally well.

In the fourth part of the questionnaire, three statements were presented to the students concerning general issues:

1. I understood the tasks.
2. I am interested in interactive learning systems.
3. I would recommend this software to other people.

Again, there were no significant differences in the responses to these three statements. The independent two-sample t-test provided the following values for item 1: t = 1.81, p = 7.5 %; for item 2: t = 0.42, p = 68%; and for item 3: t = 1.36, p = 18%. Concerning Statement 1, the results of the evaluation showed a significant (t = 43.23, p < 0.1%) mean M = 4.20 (Med = 4, SD = 0.79). This mean value indicates that every student understood the exercises—creating the keyboards and solving the intruder problem. This is important when discussing the results of the intruder problem exercise. Without this evidence that all students understood the task, the result would be meaningless.

Concerning Statement 2, the results of the evaluation showed a significant (t = 28.37, p < 0.1%) mean M = 3.91 (Med = 4, SD = 1.12). This mean value shows that most students are interested in using such tools as CoBo and UIEditor for learning complex algorithms. This is strengthened by the results of the evaluation of Statement 3, which produced a significant (t = 31.578, p < 0.1%) mean M = 3.68 (Med = 4, SD = 0.93), showing that the students would recommend the software as well as the learning method to other students.

Hypothesis 1: Cooperative/autonomous creation of concept keyboards supports the understanding of cryptographic protocols and enables the solving of more complex questions.

This hypothesis was separately analyzed for the cooperative and the non-cooperative group because the two independent-sample t-tests showed a significant (t = 7.10, p < 0.1%) difference between the two groups. Six statements were used to evaluate the hypothesis. Three of them were adapted to the groups to reflect cooperation or non-cooperation. This adaption was done in such a way that the meaning of each statement stayed the same. For example, the statement in the questionnaire for the cooperative group was 'The opportunity for discussion contributed to my motivation to solve the task'. It was changed for the non-cooperative group to 'Working individually improved my motivation to solve the task'. Both statements have the same meaning: The working style motivates the student to work, which reflects the hypothesis.

Two separate one-sample t-tests of the cooperative (t = 44.11, p < 0.1%) and of the non-cooperative group (t = 32.78, p < 0.1%) showed significant mean-values that were totally different. The cooperative group agreed with the statement with a mean value M = 3.68 (Med = 4, SD = 1.16). The non-cooperative group weakly disagreed with a mean of M = 2.82 (Med = 3, SD = 1.33). Thus, it can be seen that the cooperative group found the software more helpful than did the non-cooperative group. But in the non-cooperative group there was still no strong disagreement concerning the hypothesis.

A possible reason for this outcome is the collaboration in the cooperative group. Both groups had more or less the same pre-knowledge, which was pretty low (cf. the self-assessment). In the cooperative group, people used the opportunity to discuss ideas and impressions and to complete individuals' memories of the introductory presentation by combining the information remembered by every individual in the group. Another reason might be that it is easier to learn to use software if people consult with each other on how to use it instead of sitting alone in front of it.

Hypothesis 2: The iterative creation process motivates the learning of cryptographic protocols.

The statement 'I enjoyed the iterative construction of the keyboards' dealt with this hypothesis. The comparison of the two groups using the two independent-sample t-test showed a significant difference between the groups with $t = 2.813$ and $p = 0.7\%$. A separated analysis showed a mean value $M = 4.16$ (Med = 4, SD = 0.81) for the cooperative and $M = 3.47$ (Med = 4, SD = 1.13) for the non-cooperative group (both highly significant with $t = 29.11$ and $t = 17.84$; $p < 0.1\%$). The cooperative group thus demonstrated higher motivation. Of course, the non-cooperative group still indicated positive motivation. These phenomena can be explained by the way the cooperative group worked. Often students motivated each other, arguing about details of the protocol and trying to find a solution. Members of the non-cooperative group often had to wait for others to simulate the protocol, which possibly contributed to the lower motivation levels. But interactive computer-aided learning software tools in single scenarios were also found to be motivating, as shown in a previous study [248].

A closer look at a statement on the questionnaire designed to evaluate Hypothesis 1 shows interesting results for this hypothesis as well. Responses to the statement 'The opportunity for discussion contributed my motivation to solve the task / working individually improved my motivation to solve the task' validates the above results. The two independent-sample t-test shows that both groups are independent ($t = 2.57$, $p = 1.2\%$). The cooperative group shows a significant ($t = 32.11$, $p < 0.1\%$) mean $M = 4.19$ (Med = 4, SD = 0.74) and the non-cooperative group a mean $M = 3.5$ (Med = 4, SD = 1.33, $t = 15.33$, $p < 0.1\%$). This contributes to the assumed reasons (see the above discussion) for a higher motivation in the cooperative group as well as showing that the iterative construction is better to use in a cooperative scenario than in the non-cooperative one it was initially developed for.

Hypothesis 3: Interaction with the software to create the keyboards is intuitive and simple to learn.

Six items contributed to this hypothesis. The two independent-sample t-test indicated that both groups are significantly different ($t = 3.65$, $p < 0.1\%$). Thus, each group must be analyzed separately. The cooperative group shows a significant ($t = 50.04$, $p < 0.1\%$) mean $M = 3.98$ (Med = 4, SD = 1.10), and the non-cooperative group somewhat less strong but still highly significant ($t = 41.87$, $p < 0.1\%$) agreement of $M = 3.55$ (Med = 4, SD = 1.21). In brief, both groups agree with the statement, and, thus, the hypothesis is valid.

The possible reason for this (significant) difference is that the procedure whereby students explained to each other how to use the software was not possible in the non-cooperative group. Thus, the students working cooperatively had fewer problems using the software than the students in the non-cooperative group. Another possible reason is that, in the non-cooperative group, every student had to understand how to use the software. This was not the case in

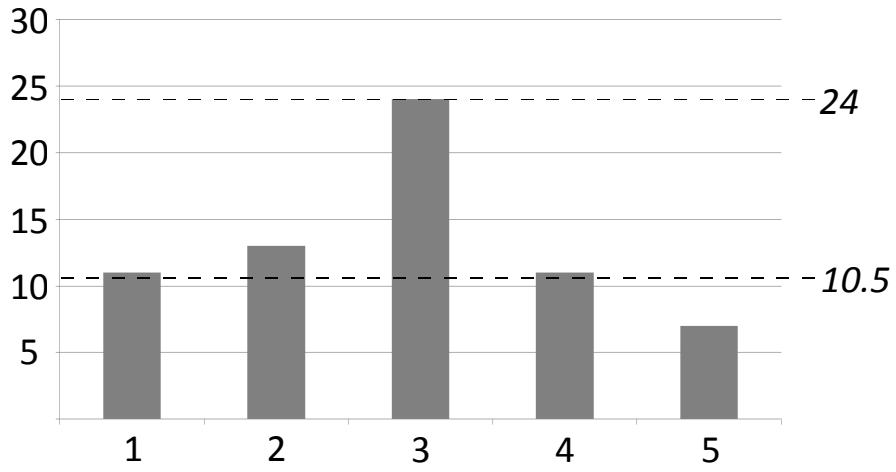


Figure 6.8.: Evaluation of learning methods

the cooperative group, where, typically, one student used the interface to interact with the UIEditor, and the other students helped him. Again, knowledge-sharing concerning the use of the UIEditor was possible in this setting but was not possible in the non-cooperative group.

On the other hand, the result from the non-cooperative group is a stronger validation of the hypothesis than the result from the cooperative group for the same reason. The students in the non-cooperative group did not have a chance to ask anyone but nevertheless found the interface intuitive (indicated by an average rating higher than 3).

Hypothesis 4: The iterative creation process is preferred to other approaches.

Four items contributed to this hypothesis. The independent sample t-test did not show any significant ($t = 0.24$, $p = 81\%$) difference between the results from the cooperative and those from the non-cooperative group. Thus, the results can be analyzed as one sample. The result is a mean of $M = 3.28$ ($Med = 3$, $SD = 1.08$). A one-sample t-test showed that this M is highly significant with $t = 49$ and $p < 0.1\%$. Concerning this mean value, there is no clear statement concerning agreement or disagreement.

In the context of this hypothesis, we asked the students to rank five different learning methods from 'I like using it' to 'I don't like using it'. The alternatives were (1) explanation of the learning material by the teacher in a classroom with a chalkboard, (2) observing the protocol running at the teacher's computer, (3) interactive creation and simulation of the steps by yourself (the presented and evaluated approach), (4) stepwise simulation of the protocol as an applet on a web browser, and (5) textual book or internet material. The evaluation of all ratings of the first position, 'I like using it', indicate that the alternatives 1, 2, 4, and 5 are uniformly distributed, which is shown by a Chi^2-test with a $Chi = 1.81$. Introducing Option 3 to the evaluation the Chi value changes it to $Chi=12.48$, which means that the distribution is not uniform. The results are shown in Figure 6.8. Option 3 is ranked in the highest position much more often than the other options. While the other options are ranked highest with a mean rate of 10.5 times, Option 3 receives the highest rating 24 times.

There is also a strong agreement in both groups concerning the statement 'I think that the creation of the keyboard, followed by a simulation and repetition when an error occurs makes sense'. The two independent-sample t-test showed that there is no significant difference between

the answers of the cooperative and the non-cooperative group. Putting the results together in one variable results in a mean of M = 4.05 (Med = 4, SD = 0.77), which is a highly significant result (t = 42, p < 0.1%).

Evaluating the statement 'I would recommend this software to other people' provides a significant (t = 31.578, p < 0.1%) mean M = 3.68 (Med = 4, SD = 0.93), which also strengthens the validation of Hypothesis 4. The results described argue for a validation of Hypothesis 4.

6.5. Conclusion

This section described a new approach to algorithm visualization and animation using concept keyboards in an extended learning process with cooperation and a thorough evaluation of the cooperative interface creation process. Positive results from former evaluations of the use of CKs in interactive learning activities motivated an extension of a distributed learning environment and complex process contexts using cooperative techniques for the creation and reconfiguration of CKs. Cryptographic protocols or algorithms are, in general, strongly specified and formally described constructs that rarely leave room for creative problem solving like real-life processes. It seems to be feasible that less strictly specified or informally described problems would offer a better scope for using flexible interfaces with adaptive interaction logic. In [282] some ideas concerning nondeterministic dynamic systems in combination with interface design issues are proposed as future work.

The results show that cooperation paired with reconfiguration of user interfaces enables users to build correct mental models of cryptographic protocols. Thus, hypothesis (a), proposed at the beginning of this chapter, which states that interaction logic is a representation of a user's mental model of a learned protocol, can also be evaluated as true. The described learning process instructed the students to use reconfiguration techniques to 'build' the protocol into the interaction logic based on their individual understanding of it. Based on the successful application in many cases, correct implementations were created. Therefore, the validation step using simulation of the protocol based on the modeled user interface shows that the user's mental model is correct and, at the same time that the interaction logic accurately represents this model. Without this connection, there would not be a positive validation of the created user interface.

The positive results of the evaluation concerning the proposed interactive workflow as an example of user interface redesign and reconfiguration motivates a further extension of the approach to distributed cooperative learning as described in [288] and as a general approach to formal design, redesign and reconfiguration of user interfaces [283]. The above-described evaluation shows that such tools for interface redesign and reconfiguration are usable and readily understood by users; they also render the cognitive computational process model of complex problems in an understandable visual form. Through this visual and physical representation of the cognitive model, the learner or user in general is able to reflect his mental model in such a way that it is possible to achieve learning success. This learning process will also be suitable in any other interaction scenario and application. Through the design, redesign, and reconfiguration of a user interface, the interaction can be more efficient and less error-prone because the user has the chance to adapt his computational process model step by step to the real machine implementation.

7. Error Reduction through Reconfiguration of User Interfaces

Error reduction was one of the primary motivations for formally modeling, reconfiguring, and redesigning user interfaces. The upcoming sections will describe two evaluation studies that investigated user reconfiguration of existing user interfaces. After introducing some basic objectives (Section 7.1) followed by a description of basic concepts and models from psychology (Section 7.2), a case study dealing with reconfiguration of input interaction processes will be examined (Section 7.3). Here, a simplified simulation of a steam-water reactor was embedded in a scenario of controlling the process of generating electricity and handling system errors as they occurred. This evaluation study was investigated in cooperation with Burkolter and Kluge as described in [284, 285]. Next, an evaluation study simulating a chocolate manufacturing process was used to investigate the reconfiguration and redesign of the output parts of a user interface, representing mostly the system state of a machine or process (Section 7.4); this study was developed and conducted primarily by Kohls and Berdys [138].

7.1. Introduction

The primary goal of HCI research is to increase the usability of user interfaces and thus to decrease the amount of human error in interaction. An important hypothesis motivating this dissertation is that individualizing and adapting user interfaces will result in a reduction of human error in interaction. This presumption has been investigated by others, such as Schneider-Hufschmidt et al. [243], who looked at the use of adaptive user interfaces. Furthermore, it has been shown above that the use of reconfiguration techniques is able to embed the user's mental model into formal user interfaces as interaction logic. The notion of attendant error reduction is based on research in cognitive psychology, which identifies the influence of mental models on human error. By approximating the interaction logic—and thereby the observable behavior of a system to be controlled—to the user's mental model, it should be possible to reduce errors in interaction. This approximation is implemented when a user applies reconfiguration to the initial state of a user interface, thus individualizing that user interface to match his or her mental model.

In HCI research, different ways of individualizing user interfaces have been described from various perspectives. Here, individualization is associated with adaptive user interfaces in the sense that adaption is implemented by formal reconfiguration of interaction logic paired with redesign of the physical representation of a user interface. This reconfiguration and redesign is applied by the individual user.

In addition to this technical perspective, it is important to take human factors into account. Here, the main hypothesis is that individualizing user interfaces through adaption by the individual user will result in better understanding and mental representation of the process to be controlled on the part of that user, thus reducing the probability of human error. Wickens and

Hollands [292] describe to what extent human factors influence the performance of a user in interaction.

To investigate the influence of user interface reconfiguration and redesign on human error in interaction, two different evaluation studies have been developed and conducted. One examined the influence of reconfiguration and redesign of parts of a given user interface for input data, and the other looked at the influence of changing parts of the user interface for output information. To this end, the initial user interfaces were created as users would expect—offering good usability. Furthermore, the physical representation was designed to give insight into the process to be controlled. The interaction logic was modeled as simply as possible, mainly redirecting incoming events to the system or sending data emitted by the system to the physical representation. When reconfiguration was applied by the user, it became more complex and more highly individualized.

For both evaluation studies, the UIEditor framework was used for modeling, simulation, and reconfiguration. The UIEditor's logging system was used to create logs that were then analyzed for errors in interaction and to evaluate users' performance while they completed several tasks with the system and their individual user interfaces.

Before describing the two evaluation studies, it is necessary to explore some basic principles that provide a solid foundation for the studies from a psychological perspective. Then, the studies and their structure will be described in detail. Finally, the results and their interpretations will be discussed. It will be shown that reconfiguration of the input and output parts of a user interface and thus an adaption of user interfaces to users' expectations and mental models results in a reduction of human error.

7.2. Psychological Principles

From a psychological point of view, the individualization resulting from reconfiguration and redesign conducted by the user himself on an initial user interface can evolve positively. Individualization puts the focus on the user, such that the user interface ultimately corresponds to the individual needs of the user. In adaptive user interface and usability research, traditional approaches in HCI research as introduced in Chapter 2 are based on designs that target generally defined human capacities, thus ignoring the individual capabilities and disabilities of individual users. Individualization of technical systems to individual real users as opposed to a generalized ideal (expert) user is thus thought to enhance performance in the human-machine or human-computer relationship, as described by Hancock et al. [103, 104]. Further positive effects on human performance are shown by Miller and Parasuraman [171], where they describe the positive effect of flexibility in human-adaptable automation.

Mental models, as described in Chapter 2, also play a central role in the study of human error in interaction with complex processes. Wickens and Hollands point out that the user's mental model "forms the basis for understanding the system, predicting its future behavior, and controlling its actions" [292]. Reconfiguration of a user interface by the individual user adapts the interface to that specific user's mental model. This means that, by reconfiguring the user interface, the user implements his or her understanding of the system directly into it. This compatibility in the interaction between user and user interface will result in a fewer errors than will interaction between a user and an unindividualized user interface. This is supported by the work of Wickens and Hollands, who state that successful performance is based on a good mental model of a system. Thus, if the user's mental model is directly implemented into the

user interface, it should decrease any differences between the mental model and the control of the system. This will be further investigated below.

Another psychological concept of interest here is how errors occur. A mental model is only one element in the complex process of information processing. A general approach to cognition was sketched in Chapter 2, Figure 2.1, and is specified by the concept of situation awareness (SA) introduced by Endsley [83]. SA refers to the perception and comprehension of information and the projection of future system states. Poor situation awareness can increase the probability of occurrence of human errors in interaction because of the lower rate of perceived information and the resulting wrong assumptions regarding future system states. With respect to HCI, the design and logic of user interfaces greatly influence SA because they determine how much information can be acquired, how accurate it is, and how correctly this information will be interpreted. Correct interpretation is greatly influenced by the user's expectation concerning the behavior of the user interface and of the system as a whole.

Based on these considerations, it is assumed that user interface reconfiguration reduces human error since it allows users to reconfigure the user interface according to their SA needs, thus reducing mental workload and the probability of error. These issues were investigated in two evaluation studies performed at the University of Duisburg-Essen in 2010.

7.3. Input Reconfiguration of User Interfaces

The first investigation sought to identify the effects of reconfiguring the input elements of a user interface on the number of errors in interaction that occurred while the user interface was being used to control a complex process and react to problems as they arose. A process was selected that is simple enough to be learned and understood in a short period of time and complex enough to make differences visible between users who reconfigured their user interface and those who did not. It was decided to choose the feedwater steam circuit of a nuclear power plant. In this circuit, pumps transport feedwater into the reactor tank, where the water is evaporated by the thermal energy emitted by the nuclear chain reaction of the nuclear fuel. This steam is transported by pipes to the turbine, where the thermal energy is transformed to kinetic energy. The kinetic energy is finally transformed to electrical energy in a connected generator. The cooled steam is then transported to the condenser, where a pump transports cooling water through pipes, condensing the cold steam to water. This feedwater is again transported by the feedwater pumps to the reactor tank, and so forth. In Figure 7.1, the whole circuit can be seen as part of the initial user interface that was given to the test persons in the evaluation study. Furthermore, this representation was used to continuously show the current system state of the feed-water circuit. Coincidentally, this part of the user interface could not be modified by the test persons because it was not part of the modeled user interface in the UIEditor framework.

The test persons were commissioned to act as reactor operators with various options for controlling the simulated feedwater circuit:

Pump speed Pump speed influences the amount of water pumped either from the condenser into the reactor core or through the condenser. This influences how much steam can be condensed back to fluid feedwater.

Control rod position The position of the control rods in the reactor defines the amount of thermal energy that is produced by the nuclear chain reaction in the core. When

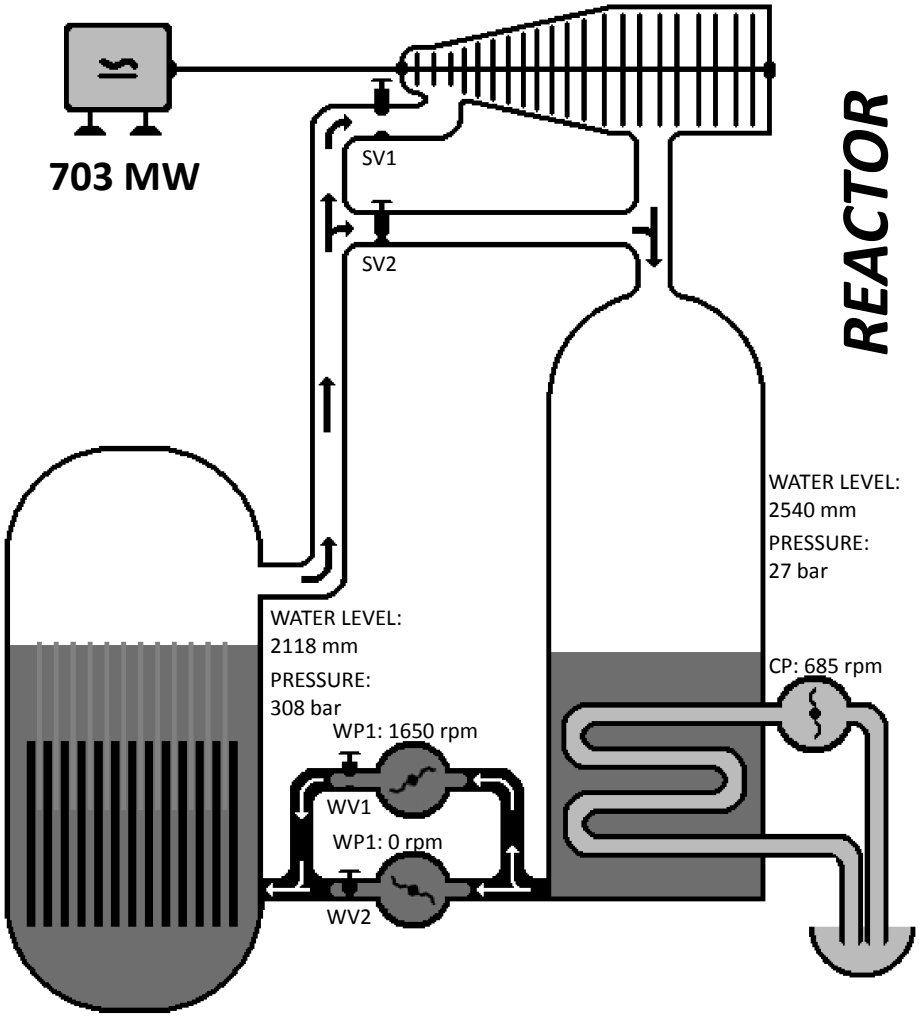


Figure 7.1.: Feed-water circuit of a pressurized nuclear power plant

the control rods are removed, more thermal energy is generated and thus more steam is generated, resulting in a higher amount of electrical energy being produced in the generator (cf. the upper portion of Figure 7.1). If the control rods are pushed completely into the core, no thermal energy will be produced and the production of electricity stops.

Valves The operator can control two sets of valves: feedwater valves (labeled *WV1* and *WV2*) and steam valves (labeled *SV1* and *SV2*).

The main task of a reactor operator is to generate the maximum amount of electrical energy while maintaining the reactor in a safe state. Here, the reactor is in a safe state if the water level in the reactor tank is at around $2100mm$ and the pressure is at around $300bar$. Thus, Figure 7.1 shows a safe state with a maximum output at around $700MW$. To hold the reactor stable, the operator has to control and observe two critical parameters: (a) the water level in the reactor tank and (b) the pressure. Both parameters can be controlled directly or indirectly by the control rods and the generated steam, as well as by the speed of the feedwater pumps controlling the amount of feedwater pumped into the reactor tank.

The process was simulated using the UIEditor framework including a simple Java implementation based on an implementation by Eriksson [86]. The initial user interface for controlling the

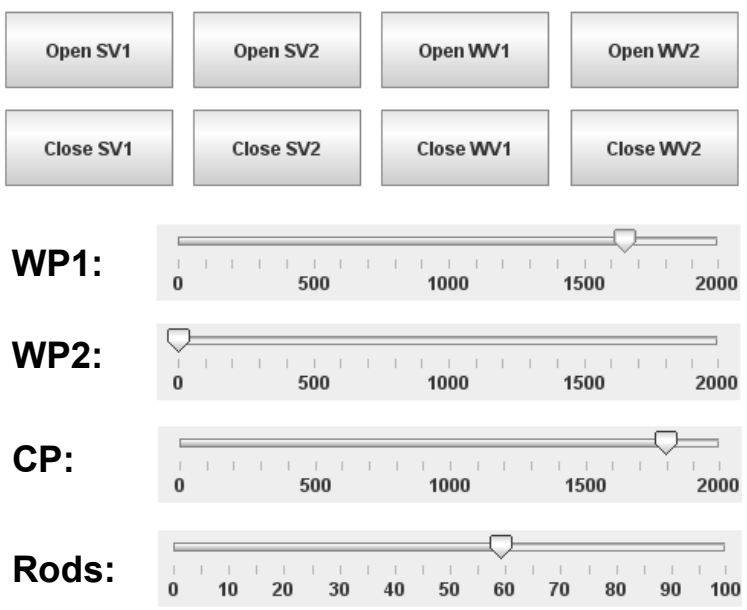


Figure 7.2.: Input interface for controlling the feed-water circuit of a nuclear power plant

process has been further modeled using the visual editors of the UIEditor framework to create the physical representation and interaction logic as well. Figure 7.2 shows the user interface for input control-information that was combined with the system visualization shown in Figure 7.1. Buttons are used to open or close the various valves, and sliders are used to set the number of rounds per minutes for the pumps and to define the position of the control rods in the nuclear core.

Next, the evaluation study will be described and the analysis of log data presented. Finally, the results will be interpreted.

Evaluation and Results

The following results and explanations were previously published by Weyers et al. [285]. It has been decided to stay close to the original work to avoid loss of information and maintain conformity with the published results.

For the accomplishment of the evaluation study, participants were randomly allocated to either the reconfiguration group (Rec-Group, n = 38) and the non-reconfiguration group (NonRec-Group, n = 34). The Rec-Group was given the ability to reconfigure their user interfaces, while the NonRec-Group used the user interface provided, which is shown in Figures 7.1 and 7.2. Thus, the NonRec-Group served as a control group.

Prior knowledge about nuclear power plants was controlled by asking the participants in both groups to give a subjective assessment of their knowledge on a seven-point scale—ranging from (1) poor to (7) very good—as well as completing a multiple-choice knowledge test with seven questions about nuclear power plants (max. 14 pts). There were no significant differences in subjective assessment of pre-test knowledge (Rec-Group: M = 4.17, SD = 1.78, NonRec-Group: M = 4.13, SD = 1.63, t(66) = 0.09, ns) or the knowledge test (Rec-Group: M = 12.38, SD = 2.17, NonRec-Group: M = 11.74, SD = 2.12, t(69) = -1.26, ns). After this pre-test, all participants received an introduction to the process and the initial interface, as well as to how

to use the interface to control the simulated feedwater circuit. Then, participants were given the opportunity to test and explore the simulator and the interface for five minutes. The Rec-Group was then introduced to the reconfiguration module of the UIEditor framework. In order to ensure the same amount of practice time and workload for all participants, the NonRec-Group was shown a sequence from a documentary about simulations. This topic was chosen as it had no direct relation to the nuclear power plant simulation, the interface, or the control task. After this phase, both groups were trained on three tasks: (1) start-up of the reactor, (2) shut-down of the reactor, and (3) handling the breakdown of the feedwater pump. All participants were provided with checklists of the procedures for every task and practiced the procedures for eight to ten minutes. After every practice session, the Rec-Group was given the time and opportunity to reconfigure their user interfaces to be better able to perform the task. During those periods, the NonRec-Group was asked to sketch ideas for possible improvements and adaptions to the user interface. Thus, both groups performed comparable tasks. This kept time on task similar for both groups. Furthermore, the sketches would help in later evaluation of the extent to which the reconfigurations applied by the Rec-Group were intuitive and what other possible reconfigurations might be welcome in addition to the restricted set offered to the Rec-Group. While the NonRec-Group was sketching ideas to improve the interface, the Rec-Group actually improved the interface using three types of reconfiguration operations: (a) duplication, (b) serialization, and (c) parallelization (cf. the descriptions in Section 4.2).

After the practice phase of the study, a testing session was held. All participants (that is, participants in both the Rec- and the NonRec-group) had to perform the following tasks: (1) starting up the reactor and dealing with the fault state they had practiced (breakdown of the feedwater pump) (2) starting up of the reactor and dealing with an unfamiliar fault state (turbine breakdown), and (3) starting up and shutting down the reactor. The NonRec-Group used the initial user interface, while the Rec-Group used their own reconfigured user interfaces without the chance to further reconfigure it during the testing session. In tasks (1) and (2), the participants did not know that the fault state would occur. During task (3), situation awareness was measured by fading out the user interface and asking specific questions concerning pressure and water level in the reactor tank. Finally, questionnaires regarding user acceptance and mental workload were filled in by the participants. The complete evaluation study can be seen in Figure 7.3.

Data Analysis

Data analysis was performed mainly on interaction logs created by the logging module of the UIEditor framework. An interaction log is a file recording all inputs made on the user interface. In a preprocessing step, the log files were reformatted to sequences of characters, each representing one specific operation. Continuous operations, like moving a slider to the right or left, were summarized to one character representing the movement of a slider to the right or to the left. Buttons controlling summarized operations in the reconfiguration group were identified by a set of characters surrounded by square brackets.

To evaluate these sequences, an expert model was created and the distance between the expert model and the test person's sequence was evaluated manually. The expert model was generated from the checklists that were handed to the test persons. Using this differentiation between expert and test person sequences made it possible to detect errors of various types, as shown in Figure 7.4.

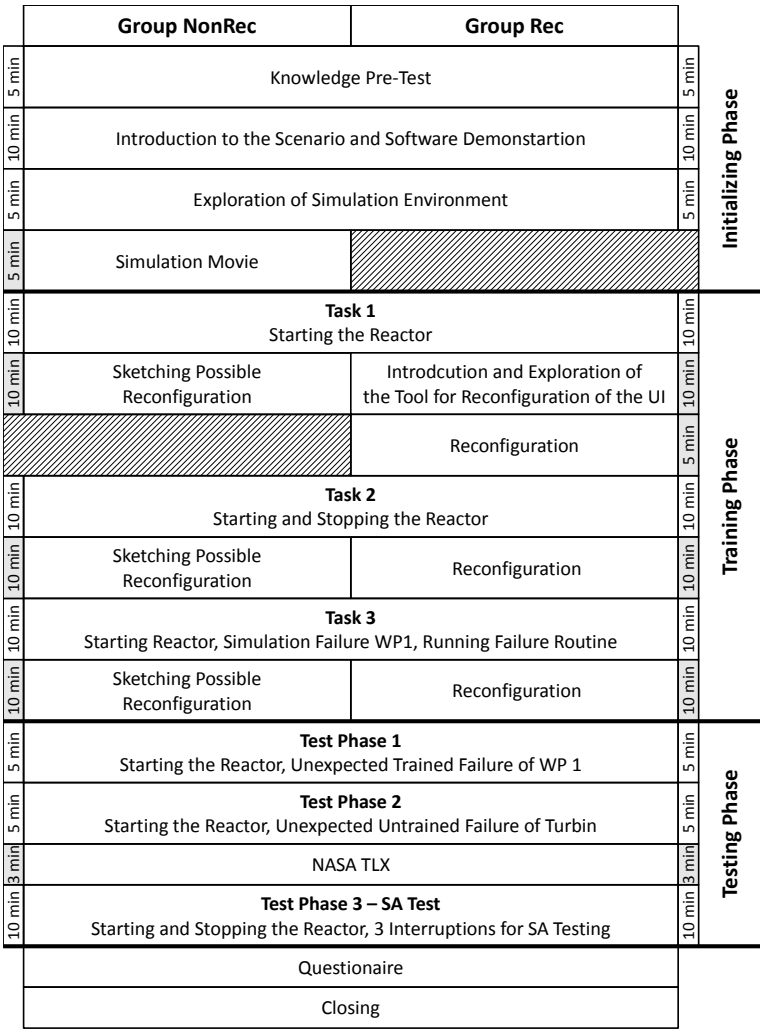


Figure 7.3.: Sequence of action in the evaluation study for investigating the influence of reconfiguring interaction processes for input

Two major groups with seven error types in total were identified, based on the error concepts of Hollnagel [113]: Magnitude errors and sequence errors. Magnitude errors are defined as movements taken too far [188]. Two types of magnitude errors were identified in the logs: oversteering and understeering. Five types of sequence errors were identified: (a) swapping two neighboring operations (reversal), (b) premature operations performed too early in the sequence (too early), (c) belated operations performed too late in the sequence (too late), (d) repeated operations that have already been executed (repetition), and (e) incorrect actions not included in the expert model (wrong actions).

Further data analysis was applied to log files generated by the feedwater circuit simulation. The control performance of the test persons was evaluated in comparison with a linear expert model shown in Figure 7.5. Meaningful parameters for such an evaluation are the water level of the reactor tank as a safety critical parameter and the power output as a command variable. The expert model in Figure 7.5 therefore shows both parameters in the three main phases of control. The first phase is the initialization phase, where various valves are opened and the cooling circuit of the condenser is started without producing electricity. In the second phase, called the start-up phase, the production of electrical energy is started before the last phase, in

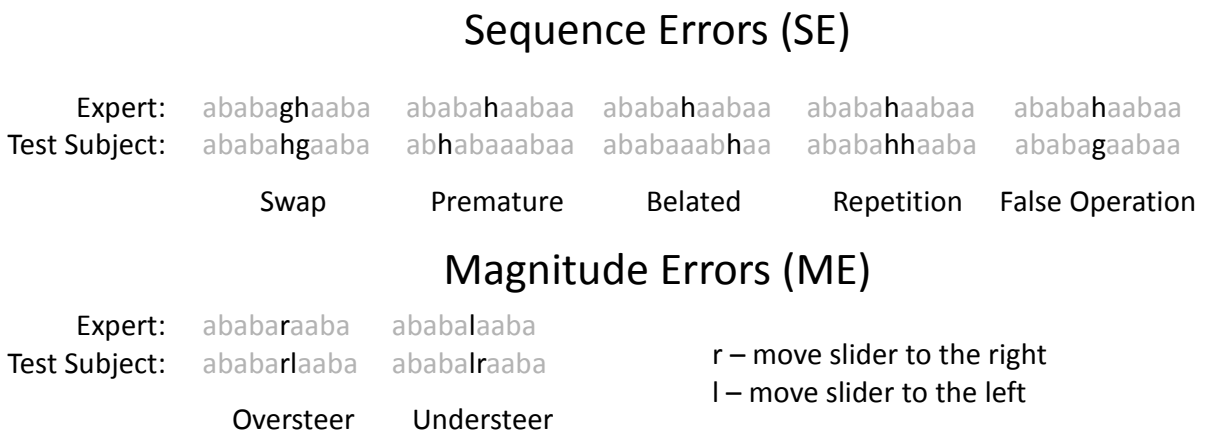


Figure 7.4.: Error types (cf. Hollnagel [113]) identified in interaction logs

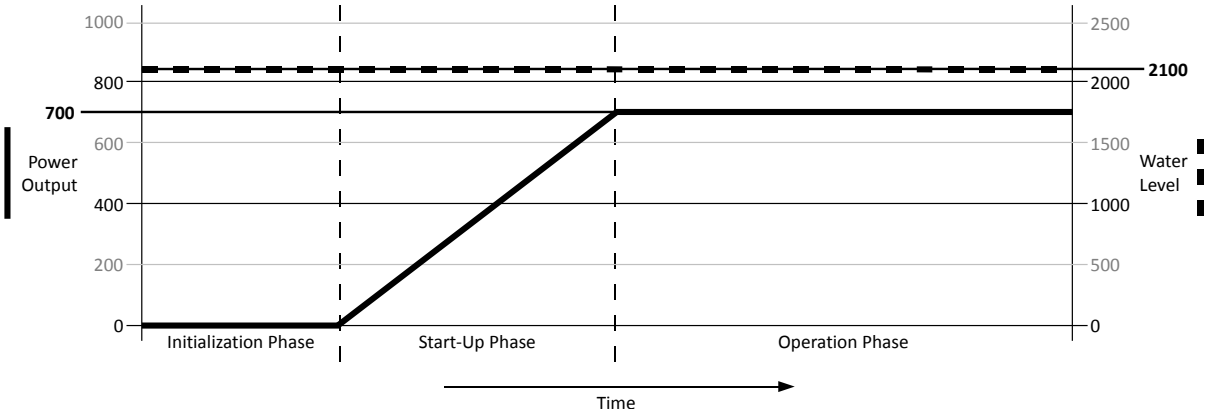


Figure 7.5.: Expert model showing an optimal initialization, start-up, and operation phase of the nuclear power plant

which the output has to be stabilized at 700 MW. For analysis, the system log files of every test person and for every task in the testing phase have been evaluated as to the periods of time used for the initialization, start-up, and operation phases.

Results

Descriptive statistics and results from t-tests for the seven error types during start-up of the reactor and the practiced fault state can be found in Table 7.1. If the assumption of homogeneity of variances was violated, degrees of freedom (df) were adjusted. Confirming our assumption, overall, the Rec-Group committed significantly fewer errors than the NonRec-Group. While the NonRec-Group committed on average 5.53 errors (SD = 2.71), the Rec-Group committed half as many errors (M = 2.48, SD = 1.93). This proved to be a large effect (r = .54), accounting for 27% of the variance [90]. With respect to the type of errors, the NonRec-Group made significantly more oversteering errors, reversal errors, and wrong actions. However, the Rec-Group committed significantly more repetition errors.

Also the results regarding start-up procedure and dealing with an unfamiliar fault are in line with our assumptions (see Table 7.2). Again, the Rec-Group committed half as many errors in total (M = 1.72, SD = 1.62) as the NonRec-Group (M = 3.38, SD = 2.32). This difference

Table 7.1.: Start-up of reactor and practiced fault state: M, SD (in parentheses) and results of t-test

Error types	Rec-Group	NonRec-group	t(df), p (one-tailed), effect size r
Oversteering	0.38 (0.70)	2.38 (1.72)	t (45.86) = -6.13, p = .000, r = .67
Understeering	0.15 (0.37)	0.15 (0.36)	t (58) = 0.07, p = .472, r = .01
Swap	0.04 (0.20)	0.47 (0.86)	t (37.40) = -2.83, p = .004, r = .42
Premature	0.23 (0.43)	0.24 (0.50)	t (58) = -0.04, p = .486, r =.01
Belated	0.42 (0.58)	0.38 (0.70)	t (58) = 0.24, p = .405, r = .03
Repetition	0.38 (0.70)	0.09 (0.29)	t (31.54) = 2.04, p = .003 , r = .34
False Operation	0.96 (1.25)	1.82 (1.90)	t (56.87) = -2.12, p = .020, r = .27
Total	2.48 (1.93)	5.53 (2.71)	t (59) = -4.93, p = .000, r = .54

Table 7.2.: Start-up of reactor and unpracticed fault state: M, SD (in parentheses) and results of t-test

Error types	Rec-Group	NonRec-group	t(df), p (one-tailed), effect size r
Oversteering	0.45 (0.83)	1.50 (1.60)	t (51.03) = -3.34, p = .001, r = .42
Understeering	0.10 (0.31)	0.18 (0.39)	t (61) = -0.82, p = .209, r = .10
Swap	0.21 (0.41)	0.29 (0.52)	t (61) = -0.73, p = .236, r = .09
Premature	0.10 (0.31)	0.38 (0.60)	t (50.82) = -2.35, p = .011, r = .31
Belated	0.48 (0.69)	0.44 (0.56)	t (61) = 0.26 , p = .396, r = .03
Repetition	0.07 (0.26)	0.00 (0.00)	t (28.0) = 1.44, p = .081, r = .26
False Operation	0.31 (0.71)	0.59 (1.02)	t (61) = -1.23, p = .111, r = .16
Total	1.72 (1.62)	3.38 (2.32)	t (61) = -3.23, p = .001, r = .38

between the two groups was significant and constitutes a medium to large effect. Regarding the types of errors, the NonRec-Group had a higher number of oversteering errors as well as sequencing errors, performing actions too early in a sequence.

Results of evaluation of the time needed for initializing and starting-up the reactor and for holding it stable after reaching the maximum output can be seen in Table 7.3. The amount of time needed for the three phases has been normalized with regard to the time needed per task. Thus, the values in Table 7.3 are percentages showing the time needed for the specific phase relative to the time needed for the complete task. Here, it can be seen that the Rec-Group needed significantly less time (M = 6.34, SD = 3.72) for the initializing phase than the NonRec-Group (M = 15.42, SD = 6.62). There is no significant difference for the start-up phase, but there is a significant difference for the operation phase, showing that the Rec-Group has more time (M = 67.07, SD = 19.67) for operation then the NonRec-Group (M = 59.19, SD = 19.59). Transforming this result into real life would mean that the test persons from the Rec-Group produce more electrical energy then the test persons from NonRec-Group in the same amount of time, resulting in more profit for the energy-producing business.

The mental workload and situation awareness of all subjects were also tested. Table 7.4 shows the results of the NASA TLX test for testing workload in the performance of a specific task. Using a questionnaire, this test makes various demands in the context of solving a given task. The test persons had to self-assess their mental and physical response to each demand on a scale from 1 (low/easy/less) to 10 (high/complex/high) after finishing the test phase in

Table 7.3.: Comparison of need time for initializing, starting up, and operating the reactor: M, SD (in parentheses) and results of t-test

Phase	Rec-Group	NonRec-group	t(df), p (one-tailed)
Initialization	6.34 (3.72)	15.42 (6.62)	t (122) = -9.56, p = .000
Start-Up	26.59 (18.95)	25.38 (16.86)	t (121.98) = .37, p = .710
Operation	67.07 (19.67)	59.19 (19.59)	t (120.08) = 2.23, p = .028
Start-Up + Operation	93.66 (3.72)	84.58 (6.62)	t (122) = 9.56, p = .000

Table 7.4.: M and SD (in parentheses) of Mental Workload Variables (NASA TLX) as a Function of Group Variables

	Rec-Group	NonRec-Group	t(df), p (one-tailed)
Mental demand: Required Activity	5.82 (1.92)	5.85 (1.91)	t(70) = 0.08, p = .468
Mental demand: Task complexity	5.03 (1.91)	5.42 (1.92)	t(69) = 0.87, p = .193
Physical demand	3.66 (2.35)	4.50 (2.84)	t(70) = 1.38, p = .085
Temporal demand	5.76 (1.70)	5.88 (1.80)	t(69) = 0.28, p = .391
Performance	4.68 (2.23)	4.29 (2.01)	t(70) = -0.78, p = .220
Effort	5.50 (1.75)	5.09 (2.29)	t(70) = -0.86, p = .196
Frustration level	6.24 (1.94)	5.68 (2.24)	t(70) = -1.14, p = .130

the evaluation study. Here, no significant differences were visible between the Rec- and the NonRec-group.

This is also true for the SA test. Here, the test persons were asked three times during the last task in the evaluation study to identify the value of a certain variable, to predict how this variable would change in the next 30 seconds (increase, decrease, stay stable), and to say how confident they were in the answers they had given. While they filled out the questionnaire, they were not able to see the user interface and the current system state. The results, which can be seen in Table 7.5, show no significant differences between the two groups. The questionnaires were evaluated by comparing the answers to the system logs created during simulation. For every correct answer, the test persons got one point. They were asked three times, and the maximum number of points per question was 3.

Table 7.5.: M and SD (in parentheses) of Levels of Situation Awareness as a Function of Group Situation Awareness

	Rec-Group	NonRec-Group	t(df), p (one-tailed)
Level 1: Perception (max. 3 pts.)	2.14 (0.92)	1.97 (0.88)	t (68) = 0.77, p = .22
Level 2 and 3: Comprehension and Prediction (max. 3 pts.)	1.54 (0.69)	1.58 (0.97)	t (57.15) = -0.18, p = .43
Mean confidence in own answers, given in percent	78.15 (13.37)	76.01 (17.02)	t (60.60) = -0.58, p = .28

Interpretation

The above-described data analysis of the 2010 evaluation study showed the strong influence of individualizing user interfaces by the user applying reconfiguration operations to an initial user interface. It was shown in a statistically significant way that reconfiguration reduces the number of errors, especially the number of errors concerning oversteering, swapping of operations, false operations and premature operations. Still, a slight increase in repetition errors was visible in the Rec-Group, which can be traced back to the newly created operations, which were sometimes erroneous and did not work well. In this case, some participants started pressing their new button several times, hoping that it would work somehow. Nevertheless, the total number of errors were reduced in a highly significant way ($p \leq .001$). Thus, reconfiguration applied by the user to a user interface reduces errors in controlling a complex process where errors of the system can occur.

Besides the reduction in the numbers of errors, performance—that is, the production of electrical energy—also increased in relation to the time needed to initialize the reactor. Still, there is no significant difference between the test and the control group concerning workload and situation awareness. Thus, it should be part of future work to investigate the influence of user interface reconfiguration on situation awareness and which psychological principles are of interest in this context.

In this evaluation study, the influence of reconfiguring user interfaces was reduced to the reconfiguration of parts of the user interface for inputting information. The output part, which represents the system state to the user, was not adapted. Thus, the next step was to investigate the influence of reconfiguring the output portion of a user interface. To that end, Kohls and Berdys [138] conducted an evaluation study very similar to the one described in this section, but with a focus on reconfiguration of output interface elements. This study will be described in the next section.

7.4. Output Reconfiguration of User Interfaces

The second investigation in context of error reduction sought to identify the influence of reconfiguring output elements of a given user interface on the number of errors in interaction while controlling a complex process combined with the reaction of test persons to upcoming problems in the process. This study was modeled on the investigation described above and conducted by Kohls and Berdys as part of their bachelor thesis [138]. The process to be controlled by the test persons was the production of chocolate. This process is subdivided into five modules, where each produces an intermediate product that is used by the downstream component as input. This sequence of modules results in a process starting with row products offered by storage and ending with the storage bearing the end product, chocolate, as can be seen in Figure 7.6. The components of this process can be described as follows:

1. **Storage** The storage offers initial products like ground cacao beans, cacao butter, milk powder, sugar and so forth, as well as storage for the end product.

2. **Mélangeur** The mélangeur kneads all initial products into a consistent mass.

3. **Roller Mill:** The roller mill mills the cacao mass produced by the mélangeur until the grain size is smaller than $25 \cdot 10^{-6} m$.

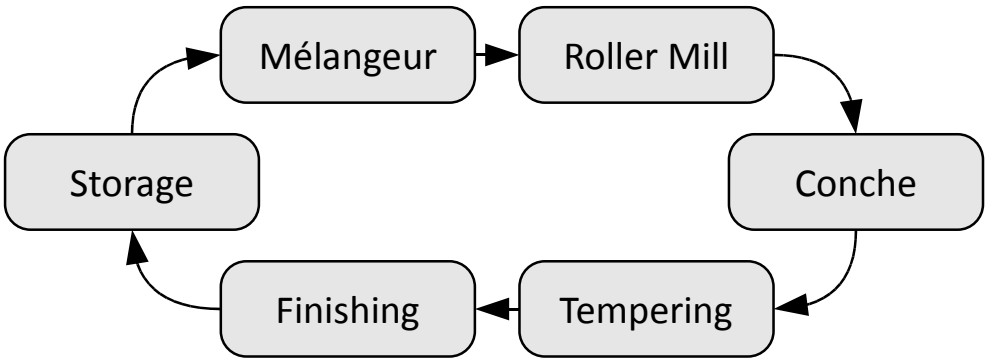


Figure 7.6.: The six stages of the chocolate production process

4. **Conche** The milled mass is then passed to the conche, where it is heated to 90°C while being constantly mixed.

5. **Tempering** After the conche has heated and mixed the cacao mass, it has to be cooled down without letting it crystallize.

6. **Finishing** Finishing handles the final portioning and molding of chocolate into bars.

In the study, for simulation reasons, various characteristics were adapted to the planned evaluation setting. For instance, the time needed for every component to finish its sub-process was reduced. In the real process, the conche requires 24 to 48 hours. This could not be transformed to the context of a study meant to last 60 to 90 minutes per cycle. Furthermore, various details were neglected in order to simplify the process so that it could be learned and understood in a short period of time. For instance, the subjects did not have to define the composition of the initial products, which in reality hardly dictates the quality of the end product.

The task for the test persons was to monitor and control certain specific characteristics and values, as well as certain parameters that were relevant to all components of the process. General parameters to be observed by the test persons were the total number of staff and the individual number of staff per component of the process. They also had to monitor parameters like the amount produced and the number of rejects resulting from faulty control of components. These might include factors like a worker entering the wrong temperature for the conche or the tempering component or the wrong number of rounds per minute for the mélangeur and the roller mill. The latter four parameters had to be checked by the test person. Faults in these areas caused problems during the chocolate scenario just as they did in the nuclear power plant scenario, when pumps broke down. In the chocolate production case, the test persons had to handle these problems, as well as changes in staff over time. Concerning staff, the test persons were able to distribute the available staff to all components, but, from time to time, some staff members went on break, which resulted in stoppages for their components. The test person had to identify this problem and stop the process or redirect working staff to the unmanned component. If the test person did not do so, the result was reject because the sub-product of the further component could only by stored for a short period of time.

Figure 7.7 shows the initial user interface that was adapted by the test group as was the case for the power plant study. The user interface shows all necessary parameters in an intuitive way. The various components of the production process are visualized and arranged in the order

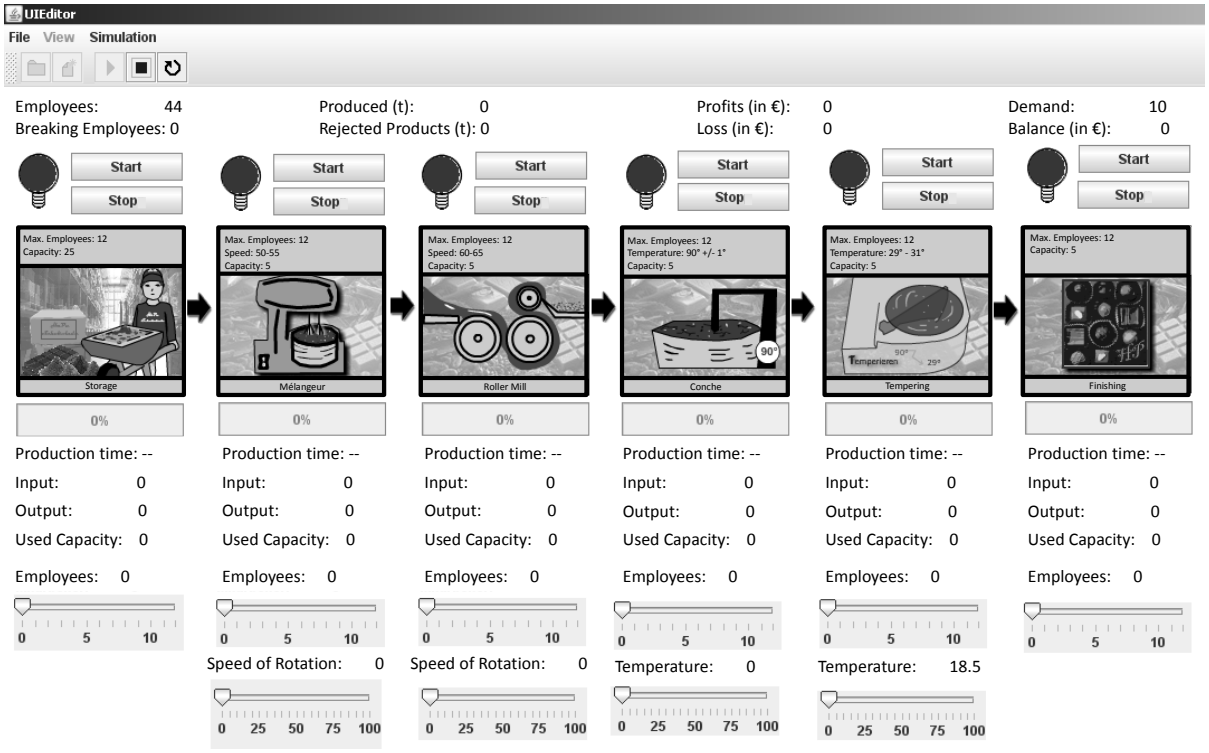


Figure 7.7.: Initial interface of the chocolate production scenario from Kohls and Berdys [138]

in which they produce the chocolate and its sub-products. Every component is equipped with start and stop buttons for starting or stopping the component, indicated by the lower image. A lamp indicates whether the component is running or not. Furthermore, every component has a slider for assigning the working staff to the associated production component. For mélangeur and roller mill, the rounds per minute can be selected using another slider. For the conche and tempering units, the temperatures are also set by slider. In the upper part of the user interface are the general parameters, such as the total number of unassigned staff, number of staff on break, and so forth.

The process logic for the chocolate factory simulation was implemented in Java programming language. The user interface was modeled, simulated, and reconfigured using the UIEditor framework. To reconfigure output elements, Kohls and Berdys developed further interaction elements [138]. Their main approach to reconfiguration was to combine several output values represented as single values in the initial user interface in a more complex multi-value representation. This representation also enables dependency visualization of certain values. This permitted not only the combination, but also the visual representation of single values instead of just text and numbers. To do this, they developed and implemented the interaction elements shown in Figure 7.8, focusing on two goals. The first goal was to represent a single value in a more comprehensive way. For instance, a temperature is visualized as a thermometer and a speed as a speedometer. Furthermore, the interaction elements they developed showed various intervals of good or bad values. Thus, the test person was able to predefine a certain interval for, say, a temperature where the thermometer image is green; for values over or under the specified interval, it turns red. The second goal was to offer a handy way to visualize a set of values in relation to one another using bar or line graphs. This kind of visualization offers a suitable way to compare different values and a quick overview of their interrelationship. For

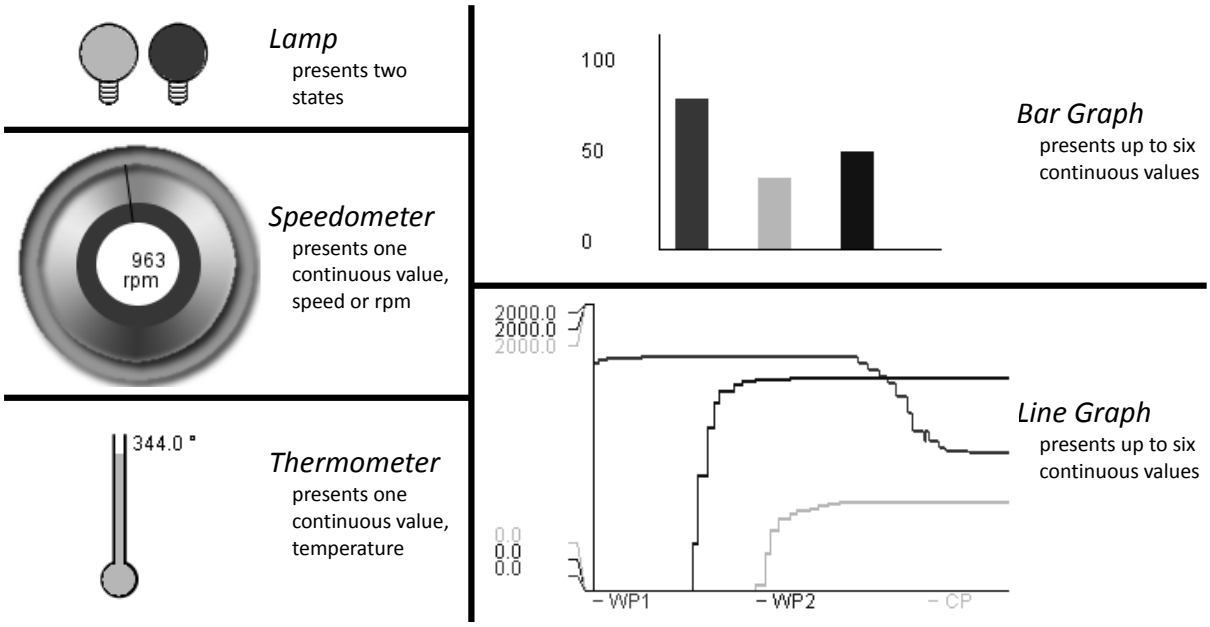


Figure 7.8.: Five more or less complex interaction elements for representing up to six continuous values in relation to one another

instance, some tasks in the study were meant to produce a predefined amount of chocolate with less waste. For these tasks, it was helpful to visualize the amount of chocolate produced in relation to waste.

The following sections describe the study and the data gathered. An interpretation of the results will follow.

Evaluation and Results

Study participants were randomly allocated to either the reconfiguration group (Rec-Group, n = 29) or the non-reconfiguration group (NonRec-Group, n = 30). The Rec-Group was given the chance to reconfigure their user interface, while the NonRec-Group used the initial user interface shown in Figure 7.7. Thus, the NonRec-Group served as a control group. The participants were between 18 and 29 years old and most were students (88%).

Prior knowledge about chocolate production was tested for both groups. The test had two sections, one dealing with prior knowledge of chocolate production and the other with prior knowledge of personal computing. The main portion of the test dealing with chocolate production consisted of seven multiple choice items with four answer choices per item. It did not indicate any significant difference between the test and the control group (Rec-Group: M = 4.17, SD = 1.42, NonRec-Group: M = 4.13, SD = 0.94, t(59) = 0.090, ns). A further item asking the participants to assess their pre-knowledge concerning the production of chocolate on a seven-point scale from 1 (very good) to 7 (very bad). This item yielded a slightly significant difference, but the arithmetic averages were very close (Rec-Group: M = 5.38, SD = 1.40, NonRec-Group: M = 4.63, SD = 1.40, t(59) = 0.045, s). A short questionnaire examined the participants prior knowledge relating to the use of personal computers. Again, a seven-point scale was used, from 3 (completely agree) to -3 (completely disagree). The results showed no significant difference between the Rec-Group and the NonRec-Group (Rec-Group: M = 0.10,

SD = 1.31, NonRec-Group: M = -0.04, SD = 1.52, t(59) = 0.702, ns).

After the participants had filled out the test and the questionnaire, they received an introduction to the chocolate production process and to the whole process for conducting the study. They were asked to put themselves in the position of the shift supervisor who controls both the equipment and the disposition of staff. As supervisor, they would also have to identify errors in the process and in staff break scheduling. After identifying a problem, they had to take measures to solve it.

Following the introduction, the participants were able to explore the system and the user interface in a guided exploration involving instructions given by the study supervisor. Next, the participants had to control the process on their own and fulfill a given task. The study was split into the equivalent of five days of production. There were therefore five test runs. Each test run had a different goal and a different type of failure in the process to be handled. Some of these failures were introduced in advance, and some were not. Furthermore, the goals of succeeding days became more and more ambitious. To permit a smooth progression of the study, each student was equipped with instructions describing the whole scenario and the various runs with their attendant goals.

After finishing the first day, the Rec-Group received training on how to reconfigure the user interface and how these reconfiguration operations could be applied to the user interface using the UIEditor framework. To keep the process equivalent for the two groups, the NonRec-Group watched a short movie during this time. Afterwards, the Rec-Group had time to apply certain reconfigurations to their interfaces, while the NonRec-Group sketched some ideas for possible adaptions of the interface on paper as done in the study described above. After the second day and the second run of reconfiguration and sketches were finished, the test phase started without further opportunity to reconfigure the interfaces. Thus, the Rec-Group used their reconfigured user interfaces, while the NonRec-Group continued to use the initial user interface. The following two days involved both groups coping with various system failures, and a fifth day was used to test situation awareness. After the last day was finished, the participants filled out two questionnaires: one testing workload using the NASA TLX test and another testing the study's hypotheses, which were similar to those in the nuclear reactor study described above. The whole process for the chocolate production study can be seen in Figure 7.9.

Data Analysis and Results

Various data sources were used to acquire date for analysis: (a) the system log files, (b) the SA test questionnaires, (c) the NASA TLX test, and (d) the follow-up questionnaire. Degrees of freedom (df) were adjusted to compensate for any violation of the assumption of homogeneity of variances.

In evaluating the log files (a), the main goal was to identify the performance of participants in controlling the process, especially in cases where system failures occurred. The basic hypothesis in this context was that using a reconfigured user interface causes fewer errors in interaction than using the user interface initially provided. Four parameters were identified that could be assumed to result from errors in interaction. The first was the number of production rejects, that is, the number of chocolates produced that could not be used for sale. Figure 7.11 contains four diagrams showing the average values per group, per day, and per parameter. Diagram (i) shows the number of rejects (rejected products); here, a slight difference can be seen on day

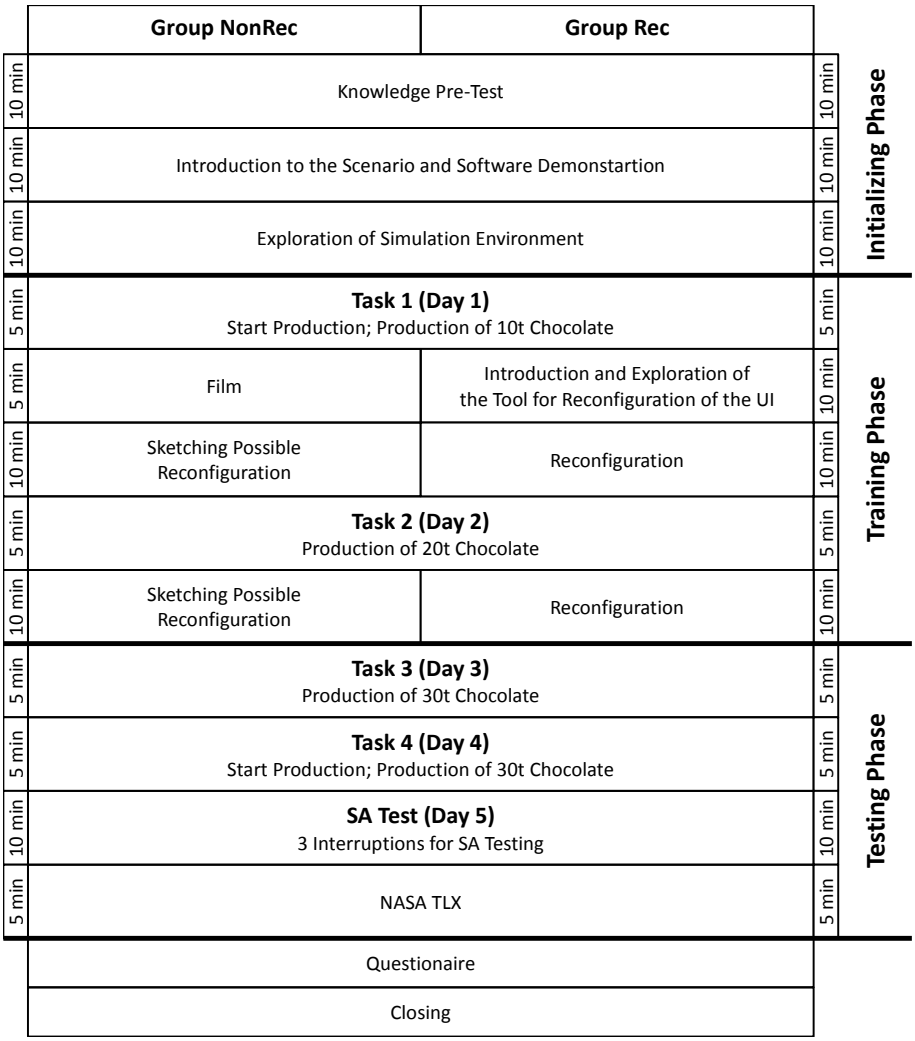


Figure 7.9.: Research procedure using a simulation of the chocolate production process

three that becomes significant on day four (Rec-Group: M = 4.66, SD = 4.81, NonRec-Group: M = 8.66, SD = 8,256, t(56) = 2.254, p = .028, s). Thus, the Rec-Group produced significantly fewer rejects on day four than the NonRec-Group. This indicates that the Rec-Group reacted more effectively to system failures in contrast with the NonRec-Group and thus made fewer errors while doing so. A further parameter was the profits from sales of produced chocolates (measured in monetary units). Here, no clear significant difference was found, but a trend can be seen in Figure 7.10: The t-values become smaller every day, reaching (t(56) = -1.739, p = .088) on day 4, when the Rec-Group has greater profits then the NonRec-Group. This trend is supported by the evaluation of loss (iii) measured in monetary units, also shown in Figure 7.11, where the loss produced by the NonRec-Group is significantly larger on day 4 than is the case for the Rec-Group (Rec-Group: M = 153620.69, SD = 158820.75, NonRec-Group: M = 285620.69, SD = 272450.05, t(56) = 2.254, p = .028, s). The combination of profits and loss resulting in the balance also supports this hypothesis. The balance is significantly higher for the Rec-Group than for the NonRec-Group (Rec-Group: M = 803379.31, SD = 158820.75, NonRec-Group: M = 625862.07, SD = 381542.43, t(56) = -2.313, p = .024, s). Thus, overall, the Rec-Group shows significantly better results on day 4 than the NonRec-Group. The Rec-Group also makes fewer errors in interaction than the NonRec-Group. Day 5 was not included in the analysis because

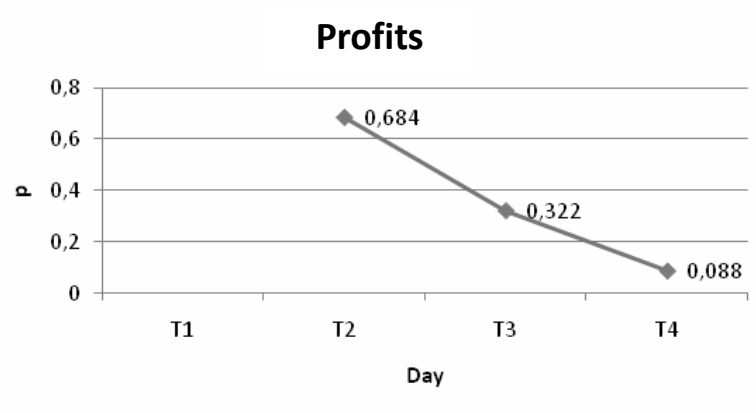


Figure 7.10.: Evolution of p value of t-test applied to log data for investigation of profits

only the SA Test was conducted on that day. All the results of the analysis for day 4 can be seen in Table 7.6.

Situation awareness (b) was evaluated using a questionnaire paired with an analysis of log files to validate answers given by the participants. Situation awareness was measured by controlling the process, which was interrupted three times. Each time, the participants were asked three questions. The first was a multiple choice question asking them to identify the current value of a given parameter that is important in the chocolate production process. The second question asked them to predict the development of that parameter over the next 30 seconds. That is, they had to decide whether the parameter would increase, decrease, or stay stable. The last question asked them to indicate how certain they were concerning the prediction they had just made. The participants' answers were compared with the real values in the log files. Correct answer received one point; incorrect answers received 0 points. Statistical analysis yielded no significant difference between the Rec-Group and the NonRec-Group, as can be seen in Table 7.7.

Table 7.8 shows the analysis of the NASA TLX test (c), which measured the perceived demand of the cognitive and physical workload. Every item was measured on a 10-point scale from 0 (low/easy/less) to 9 (high/complex/more). Only in the case of the two items for cognitive load did the Rec-Group differ significantly from the NonRec-Group ($t(57) = -3.66$, $p = .001$ and $t(56.46) = -2.47$, $p = .017$). In both cases, the Rec-Group perceived a higher cognitive load than did the NonRec-Group. A possible explanation for this is that the participants of the Rec-Group also rated the reconfiguration as part of the cognitive load, as opposed to the NonRec-Group, who did not configure the user interface. A further result worthy of note is that the perceived stress was identical for both groups. Thus, reconfiguration has less influence on this factor than on errors in interaction.

The relevant results from the general questionnaires (d) was the evaluation of the UIEditor framework used to reconfigure the interfaces. This was only rated by the Rec-Group. Three items addressed the usability of the reconfiguration tool. All three were rated on a seven-point scale from +3 (very good) to -3 (very bad). The statistical analysis showed a significantly positive result for usability of the UIEditor reconfiguration tool ($M = 1.06$, $SD = 1.40$, $t(28) = 4.07$, $p = .000$). Other results did not shown any significant differences between the Rec- and the NonRec-Group.

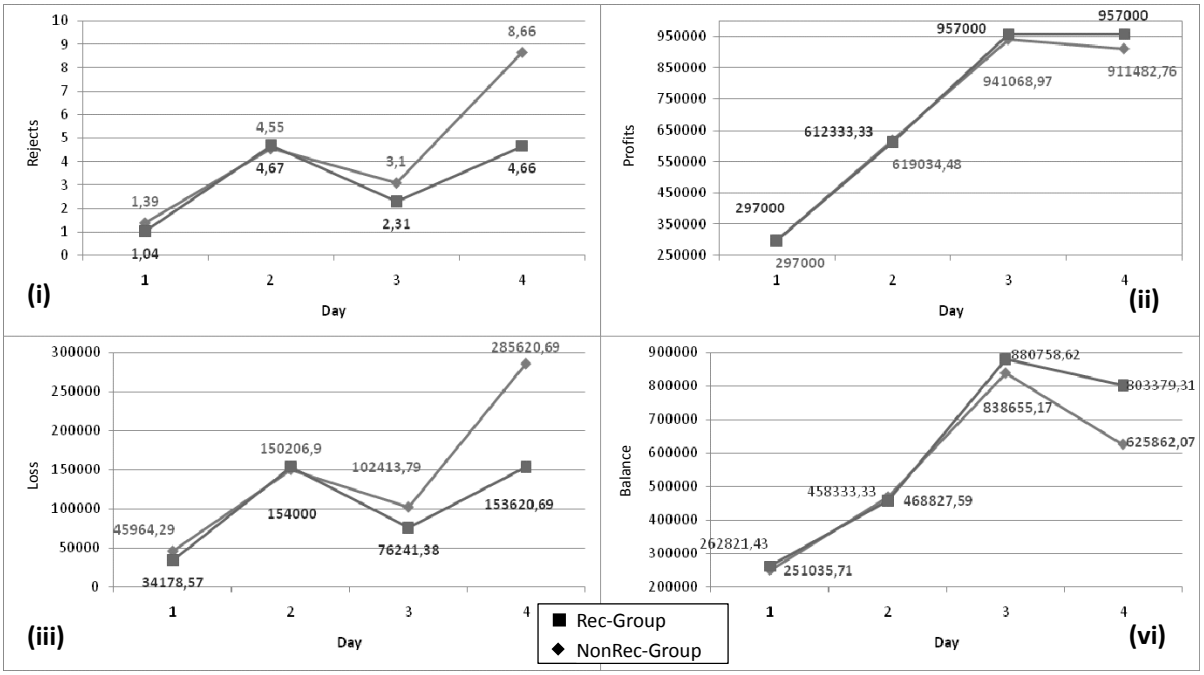


Figure 7.11.: The four parameters investigated to evaluate the performance in controlling the production process by participants separated into two groups (Rec-Group and NonRec-Group): (i) production rejects, (ii) profits, (iii) loss and the (iv) balance, or difference between profit and loss.

Table 7.6.: Participant's performance on task (day) 4: M, SD (in parentheses) and results of t-test

Performance Parameter	Rec-Group	NonRec-group	t(df), p (one-tailed)
Rejects	4.66 (4.81)	8.66 (8.26)	t(56) = 2.25, p = .028
Receipts	957000.00 (.00)	911482.76 (140952.28)	t(56) = -1.74, p = .088
Loss	153620.69 (158820.75)	285620.69 (272450.05)	t(56) = 2.25, p = .028
Balance	803379.31 (158820.75)	625862.07 (381542.43)	t(56) = -2.31, p = .024

Interpretation

As the results of the data analysis show, the reconfiguration of output interaction elements significantly influence the interaction involved in controlling a complex process. In particular, success in control can be positively influenced by an adequate and user-specific reconfiguration of output elements. For other psychological aspects, some indication of an influence was found, but this was not as clear as it was for performance while controlling the process and dealing with problems as they occurred. Thus, the SA test revealed only slight differences without statistical significance. The same is true for the NASA TLX test. The NASA TLX test showed a significantly higher cognitive load for the Rec-Group in comparison with the NonRec-Group, probably reflecting the higher workload involved in reconfiguration. In contrast, the post-test questionnaire showed a significantly positive perceived usability in using the reconfiguration tool as part of the UIEditor framework.

All in all, this evaluation study demonstrated that reconfiguring interaction elements for

Table 7.7.: Results of situation-awareness test: M, SD (in parentheses) and results of t-test

SA test	Rec-Group	NonRec-group	t(df), p (one-tailed)
No. 1	1.14 (.69)	.83 (.70)	t(56.96) = -1.68, p = .098
No. 2	1.14 (.79)	1.17 (.70)	t(55.66) = .15, p = .883
No. 3	1.66 (.61)	1.77 (.43)	t(50.01) = .81, p = .424
Total	3.93 (1.46)	3.77 (1.14)	t(52.82) = -.48, p = .632

Table 7.8.: Results of NASA TLX: M, SD (in parentheses) and results of t-test

NASA TLX	Rec-Group	NonRec-group	t(df), p (one-tailed)
Cognitive Load I	5.52 (1.45)	3.87 (1.96)	t(57) = -3.66, p = .001
Cognitive Load II	4.48 (1.77)	3.27 (2.02)	t(56.46) = -2.47, p = .017
Physical Load	1.72 (2.05)	1.97 (2.01)	t(56.83) = .46, p = .648
Time Load	5.00 (1.96)	4.27 (1.86)	t(56.53) = -1.47, p = .146
Performance	2.14 (2.33)	2.33 (2.12)	t(56.12) = .34, p = .738
Stress	3.52 (2.31)	3.53 (2.27)	t(56.85) = .03, p = .979
Frustration	5.56 (2.35)	5.37 (2.48)	t(56.98) = -.29, p = .770

output information has a promising positive influence. At the same time, the study raised further questions that should be investigated in greater detail, especially the role of situation awareness and the cognitive workload resulting from this scenario.

7.5. Conclusion

This chapter introduced two evaluation studies that investigated the influence of reconfiguration applied by the user to an initial user interface on the number of human interaction errors. A special focus lay on investigating the user's reaction to known or unknown faults in the controlled system and thus on validating the users' ability to handle this situation and their understanding of the process and the user interface.

The first study evaluated the influence of reconfiguration of part of the user interface for input data and its processing. A simulation of the feedwater circuit of a boiling water reactor was implemented and connected to the UIEditor framework, where a user interface was created for simulation. This implementation was then evaluated by comparing two groups of participants: one group that reconfigured the initial user interface, and one that did not. An evaluation of the interaction logs showed significant differences in the number of errors due to error classes defined by Hollnagel [113].

A second study, carried out by Kohls and Berdys [138], also showed the influence of this kind of reconfiguration on controlling complex processes and handling system errors. Furthermore, questions arose from this study concerning the cognitive load produced by reconfiguration and how this influences the entire interaction process.

In conclusion, individualization through reconfiguration (that is, reconfiguration by the user) applied to an initial user interface has a (significantly) positive influence on interaction, whether the input or output parts of the user interface are reconfigured. It has been shown that through reconfiguration (a) the mental model can be transferred to a representation as interaction logic and thus become a part of the user interface and (b) a positive effect can be generated by

adapting interaction logic to the user's mental model. Still, the question of what the effects would be if both input and output aspects were reconfigured remains to be investigated. Furthermore, future studies must also address the question of how having to reconfigure both input and output aspects of an initial user interface will affect a user. Other questions arose from these studies, as well. For instance, does reconfiguration increase situation awareness? In this context, further investigation of automatic and semi-automatic reconfiguration is relevant. This is because, when reconfiguring a user interface, the user will fail to identify some problems in the individual adaption of the interface, which will not affect SA positively. Therefore, if the reconfiguration system identifies problems in interaction that the user is not aware of, it can propose certain reconfiguration operations and make the user aware of unperceived issues. The following chapter will introduce some areas for future work that will extend the formal concept of reconfiguration and the implementation of the UIEditor framework with concepts and components for error-based automatic reconfiguration.

8. Automatic Reconfiguration of User Interfaces

This chapter explores opportunities for future work concerning the application of automatic reconfiguration to formal user interfaces. It begins with an introductory overview of a possible scenario in which reconfiguration can be applied to a formal user interface (Section 8.1). This overview does not cover various aspects currently being investigated, such as multi-user and multimodal user interfaces, or topics involving greater involvement of formal modeling in cognitive psychology and human-machine interaction from the perspective of control and automation. These kinds of future work aspects will be discussed in Chapter 9. After the introduction, a basic prototypical approach to a possible reconfiguration system will be described, providing a summary of the work to be done in order to support efficient reconfiguration in the scenarios described in Section 8.2. The final section in this chapter examines the work of several students who, under the author's supervision, developed and implemented approaches to interaction analysis as a central factor in generating sufficient reconfiguration rules in a variety of scenarios (Section 8.3).

8.1. Introduction

The previous chapter described and interpreted two evaluation studies involving reconfiguration. However, in both studies, reconfiguration was applied manually by the user. The next step will be to investigate automatically generated reconfiguration and redesign. Automatic reconfiguration can be described from two main perspectives: on the one hand, how reconfiguration is triggered and how much is applied, and, on the other, the extent to which the user should be involved in reconfiguration. Furthermore, the goal(s) of reconfiguring the user interface has to be taken into account: to reduce the incidence of errors or to fulfill other requirements, such as multi-user access.

In this chapter, some of these aspects will be briefly discussed. In some cases, basic work has already been carried out by various students trying to identify future questions in research and implement basic software components for further development. Here, the primary focus was on developing components to analyze interaction and provide a basis for further investigation of automatic reconfiguration. Without a solid implementation of analysis methods for interaction, it is not possible to identify errors or patterns in interaction that are key in user interface reconfiguration. Further investigation is also needed into the introduction of knowledge about the situation and environment in which the interaction is executed. Moreover, the involvement of models of the system in the reconfiguration process should be studied to offer a knowledge base for the creation of reconfiguration rules.

Figure 8.1 shows a three dimensional coordinate system in which the axes define when reconfiguration is applied to a user interface (offline, interrupt, online), who triggers the reconfiguration (user, reserved trigger by the system, exclusive the system), and how the reconfiguration is

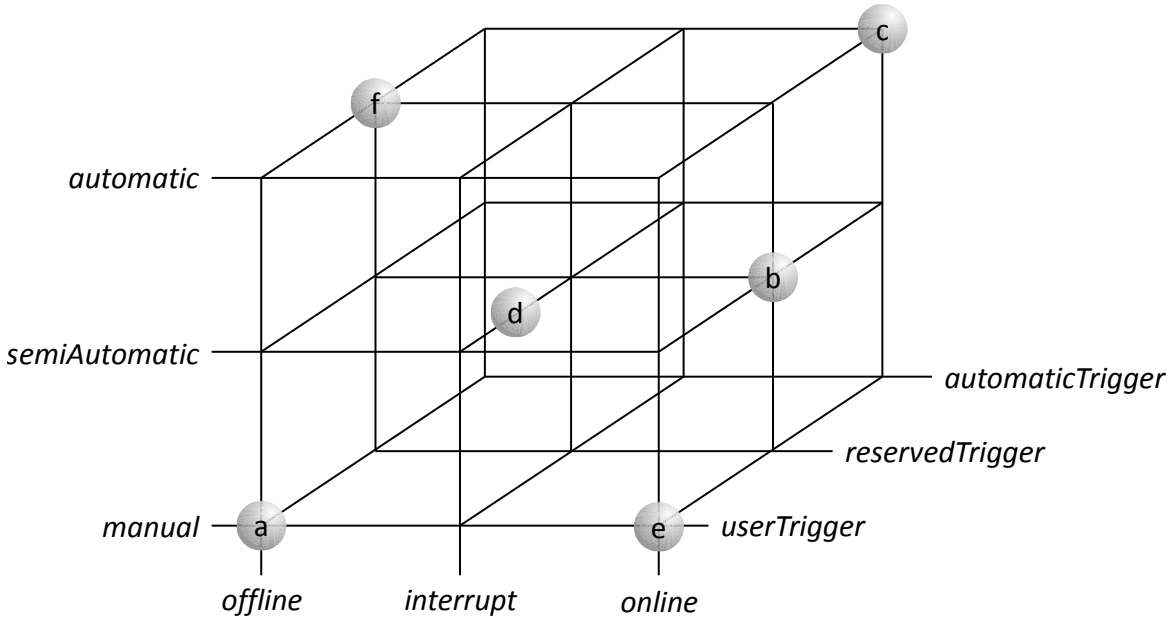


Figure 8.1.: Overview of possible future work scenarios

applied (manually, semi-automatically, automatically). Thus, every scenario is defined through a three-dimensional vector

$$\mathbf{s} \in V_1 \times V_2 \times V_3,$$

where

$$\begin{aligned} V_1 &= \{\textit{offline, interrupt, online}\}, \\ V_2 &= \{\textit{userTrigger, reservedTrigger, automaticTrigger}\}, \textit{and} \\ V_3 &= \{\textit{manual, semiAutomatic, automatic}\}. \end{aligned}$$

Based on this scenario classification, an architectural approach to a reconfiguration system involving the above-described aspects (user models, system models, various knowledge bases, and analysis) can be developed. However, before doing so, the different types of axes will be explained by the scenarios (a–f) indicated in Figure 8.1 as follows:

a=*(offline, userTrigger, manual)*: This scenario is very basic and has been used in all evaluation studies so far. Here, the interaction process is stopped completely, which also includes the (simulation of the) controlled system. This is indicated by the first value in the scenario vector: *offline*. The second value indicates whether the user or the system triggers the reconfiguration. Here, the user decided when to reconfigure the user interface indicated by the parameter value *userTrigger*. The last parameter value indicates who decides how to reconfigure the user interface. Here, again the user makes this decision. This is indicated by the parameter value *manual* as last element of the scenario vector.

b=*(online, reservedTrigger, semiAutomatic)*: In contrast to scenario **a**, this is an online scenario. Here, neither the user interface nor the controlled system for applying reconfiguration is stopped. Thus, the user is able to continue interacting with the system even while parts of the user interface are reconfigured. Furthermore, reconfiguration is triggered reservedly and applied semi-automatically. Reserved triggering means that the

reconfiguration system triggers the reconfiguration and includes the user in this decision. The system also includes the user in the definition of reconfiguration by making suggestions for possible adaptions without applying them directly to the user interface. This is indicated by the parameter value *semiAutomatic*. This scenario would apply, for example, in a real-life situation in which the system cannot be stopped because it is safety critical and the reconfiguration extends the assisting features of a user interface.

c=*(online, automaticTrigger, automatic)*: In this scenario, reconfiguration is fully automatic. It is applied automatically by the reconfiguration system without involving the user. Furthermore, the triggering of reconfiguration is controlled completely by the system, and reconfigurations are applied to the user interface during runtime, as indicated by the *online* value in the scenario vector. Here, the same safety-critical system can be taken into account with another motivation for reconfiguration. Possible objectives for implementing fully automatic reconfiguration of the user interface include limiting the inputs the user can apply in order to reduce user error and supplying certain important information to the user to steer his awareness.

d=*(interrupt, userTrigger/reservedTrigger, semiAutomatic)*: One or more values of a scenario vector can also be fuzzily defined, which is the case for the trigger parameter in this example. Here, both the system and the user are able to trigger reconfiguration. Moreover, a combination of automatic and user-triggered reconfiguration is also possible. In this scenario, the reconfiguration is semi-automatically defined; this means that, for instance, the system suggests a certain reconfiguration and the user is asked to apply the suggested reconfiguration to the user interface or cancel it. In order to apply a reconfiguration in this scenario where the system is still running, the interaction is interrupted (indicated by the parameter value *interrupt*).

e=*(online, userTrigger, manual)*: Here, the user triggers and manually applies the reconfiguration while the system and the user interface are still running (*online*). Thus, only the reconfigured part of the user interface is not usable for a short period of time during reconfiguration. This is relevant if it is important that the system remain in uninterrupted use. Furthermore, other users can still use the remaining user interface, which is important in multi-user scenarios.

f=*(offline, reservedTrigger, automatic)*: In the final scenario, the reconfiguration is applied only after stopping the interaction and the system. The user is involved in the triggering, but the system decides when to trigger the reconfiguration (*reservedTrigger*). Furthermore, reconfiguration is then applied by the system, and the user has no influence on what will be reconfigured. In a real situation, the system will ask for reconfiguration and probably indicate what will be reconfigured, but the user has no further influence on what will be reconfigured or how.

In general, this kind of classification of reconfiguration scenarios can be futher extended by adding axes that indicate other factors, such as the number of users involved, or by changing the granularity of the axes described above. The following list gives an overview of possible extensions of the classification:

Influence Axis This dimension addresses the question of who influences the system on a more general level with regard to applying reconfiguration. Direct influence, such as

triggering reconfiguration, should be separated from indirect influences. Indirect influences might, for instance, come from the environment; an example is changes in time that influence the user, the system, or both. Furthermore, direct influences could emanate from a super-user, a supervisor, or the like. Thus, influence can originate with

a) human users, who may be organized in a hierarchical command structure (for instance, controller, supervisor, chief, local directive, etc.),

b) the environment,

c) the system, and

d) an automatic source that has been implemented to influence the interaction and reconfiguration process, such as an automatic triggering system as described above.

Motivation Axis This axis defines various types of motivation for reconfiguration. Here, the following motivations can be mentioned:

Error reduction Here, the goal is to reduce errors in interaction in order to strengthen the effectiveness of interaction and reduce risks.

Change of responsibility The responsibilities of human users can change over time, which should be reflected in the user interface offering more or less access to monitor and control the system.

Change in user psychology/capabilities During interaction, the ability of the user to effectively and accurately interact with the system may change. For instance, symptoms of fatigue could arise. In such cases, the requirements of the user interface should adapt to the new situation. Thus, a user interface should be reconfigured dependent on the user's psychological abilities, which may vary over short or long time periods.

Individualization Users usually have different requirements regarding a user interface in context of a specific task and the system to be controlled. Thus, reconfiguration should be applied to the interface to adapt it to the user's needs and abilities in controlling the system.

Automation and Assistance Growing experience and former experiences on the part of a user with a given user interface or system can also result in the need for automation of certain interaction and control processes. A growing need for assistance can result, for instance, from a change in situation or environment, or from certain usage.

Task axis A user interface should be reconfigured to reflect the various characteristics that define a given task.

Context axis The context in which a task is solved also influences the demands on the user interface involved, possibly necessitating reconfiguration. Even if the same task is being done, a different context could require a completely different user interface to perform that task.

This list is probably not complete but gives an overview of the broad spectrum of future research topics that are of interest in the area of computer-supported reconfiguration of user interfaces.

The three-dimensional scenario classification described above provides an insight into the complex nature of the problem of reconfiguration and seeks to highlight important topics for further research and development. The following sections will describe a conceptual architecture for implementing a reconfiguration system that contains all the necessary components and data models. This is followed by examples of initial work in this area that addresses interaction analysis in such a way that the scenario classification can be implemented in this architecture.

8.2. Reconfiguration System Architecture

Figure 8.2 introduces a possible architecture for implementing a reconfiguration system for applying reconfiguration to a (formally modeled) user interface. Various parts of the presented system currently exist as modules in the UIEditor framework presented in Chapter 5. These are the *UI Executer*, which is, for the most part, equivalent to the *Simulator* module; the *User Interface Model* as a runtime instance of the `.uie` and `.pnml` files of the physical representation and interaction logic of the relevant user interface; the *Interaction Logger* paired with the *Interaction Logs* as output; the *Interactive Reconfiguration Editor*, which is part of the *Reconfigurer* in the UIEditor framework and indicated in Figure 8.2 by rounded corners to show that it is controllable by the user as is the user interface; and the *UI Reconfigurer*, which applies generated rules to the user interface model that was identified as a component of the *Reconfigurer* module in the UIEditor framework module structure. What is new is the *Interaction Analyzer* component. This component analyzes the interaction during runtime or offline; it involves a user model, a model of the system, and a task model that specifies what the user wants to do. The main goal of this kind of analysis is to identify abnormalities in the interaction process by analyzing the data passed through the interaction logic to and from the user or the system. Identification of abnormalities can range from the identification of recurrent input operation sequences to complex error detection involving complex context-aware modeling approaches. This module was called the *Analyzer* module in the above-described module structure of the UIEditor framework, but was not further discussed because its primary relevance is to future work. If reconfiguration seems necessary from the perspective of the interaction analyzer, reconfiguration is triggered by another new module called the *Trigger Handler*. The trigger handler controls the triggering of reconfiguration by the interaction analyzer or the user or both (depending on the scenario), as well as the stopping of the system and/or the user interface if necessary. Furthermore, beside starting the trigger handler, the interaction analyzer generates output for the *Reconfiguration Rule Generator*, who also generates rules based on predefined knowledge indicated as *Reconfiguration Patterns* in Figure 8.2. These patterns can be of various types, but in general can be seen as a source of knowledge for rule generation that is not derived from actual current interaction.

The generation of reconfiguration patterns is another problem to be tackled in future work. In general, these patterns can always be supported by experts who are familiar with user interface modeling and creation as done in this work concerning the reconfiguration operations presented in Section 4.2. This manual approach could be supported by intelligent algorithmic concepts combined with more abstract modeling of interaction operations. Thus, by assigning interaction operations to various classes, it could be possible to investigate reconfiguration operation applied by the user to given user interfaces and to derive interaction patterns by classifying the reconfiguration operations. For instance, if a user frequently combines two operations of a specific type with a given reconfiguration operation, the resulting pattern would be a more specific

reconfiguration operation to be applied only to specific types of interaction operations. Nevertheless, the creation of patterns will always need manual intervention by an expert modeler who is able, on the one hand, to understand the theoretical background of the reconfiguration system and the user interface model and, on the other, to model the reconfiguration patterns in the correct modeling language (which also needs to be investigated in future work). In general, this process of creating patterns will be an offline activity in a modeling process rather than part of the runtime reconfiguration processes. Furthermore, all the patterns generated must be proved not to create reconfiguration rules that violate the conditions introduced in Section 4.2 above. In future work, these conditions have to be tested for consistency and integrity, preventing automatically created rules from changing the interaction logic in any unwanted way.

The rules thus generated are then applied by the *UI Reconfigurer* module to the user interface model, as was the case in the UIEditor framework.

Figure 8.2 shows other elements and modules that address the extension of axes to the basic three-dimensional model for scenario classification introduced above. The hatched boxes indicate elements to be added to the architecture in order to extend it further. Thus, environment and context should be introduced as influences on the system, as well as on interaction analysis and rule creation. These kinds of models could be introduced in the reference net-based modeling approach by referencing them resulting in a hybrid model of interaction logic. Furthermore, the trigger handler should be extended by adding an interface for triggering reconfiguration that does not result from interaction analysis but from other sources, like the authority of a supervisor.

The most work remains to be done in the development of efficient approaches to interaction analysis using a user interface that generates information important for triggering reconfiguration and generating reconfiguration rules. To this end, various approaches can be considered for inclusion in future research:

Interaction Log Analysis Interaction logs can be analyzed for different purposes, for instance, to identify errors in interaction or special patterns that identify common sequences of interaction operations.

Inclusion of System and Task Models The System Model can also be included in interaction analysis. By including the system model, decisions can be made concerning the current system state and which state should be positive in order to reach a specific goal defined in a given task model. The user interface can be reconfigured such that this state can be more easily achieved. The goal to be reached is then part of the user model.

Inclusion of User Interface Model Including the user interface model offers the identification of problems involved in a (formal mental) user model of a particular user interface. Thus, problems and errors in interaction can be identified in advance or in combination with interaction data during runtime.

This list is not complete, but provides a short overview of various aspects of analysis of the interaction in this architecture. The following section will offer deeper insights into the subject of log file analysis, which has been investigated by Altunok [3], Borisov [29] and Yu [303].

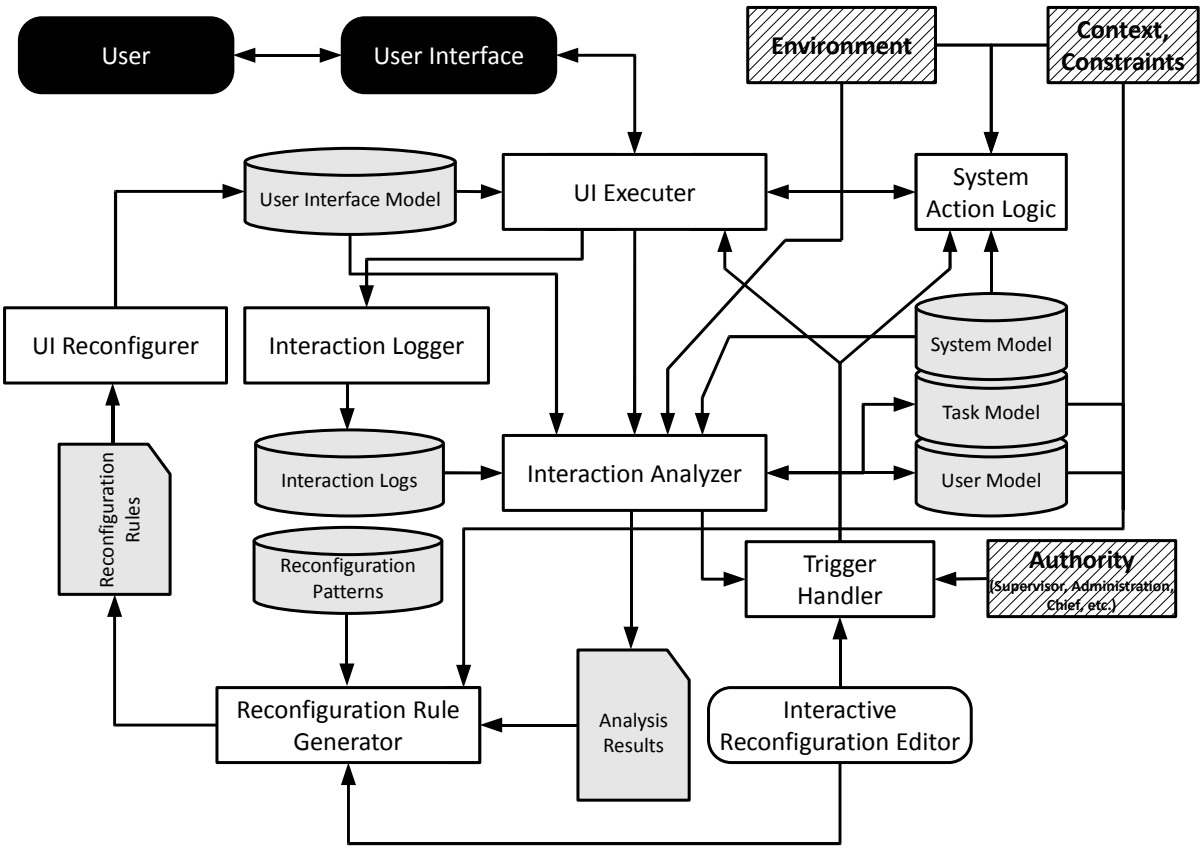


Figure 8.2.: Possible architecture of a reconfiguration system covering the scenario space presented in Section 8.1

8.3. Interaction Analysis

The first work in this area was carried out by Altunok. In his thesis [3], he implemented a software component that transforms log files into a cvs-based format based on strings, where every character represents one operation derived from the log file. The main problem was to represent continuous interaction operations with only one sign. Therefore, a continuous input operation was transferred to one sign by encoding an *increase value* or *decrease value*.

Next, the created string representing a sequence of input operations (or the triggering event of interaction processes for input data) was further analyzed using string-based algorithms for pattern matching and prediction. The goal was to identify certain patterns in a given interaction sequence using prediction algorithms and to verify the results using pattern-matching approaches. The resulting patterns would then be used in interaction as input for the generation of reconfiguration rules, for instance, to create new interaction processes that sequentialize recurring patterns or to automate certain interaction sequences that have often been applied by the user to the user interface.

For string-matching, Altunok implemented algorithms developed by Knuth et al. [137] that extend the naive brute-force approach described Ottmann et al.[198] by taking advantage of recognized parts of the searched pattern before a mismatch occurs. Based on this information, the algorithm can skip several positions instead of only one in every step, as is the case in the

brute-force approach. A further example is the algorithm developed by Boyer and Moore [32] that uses heuristics to speed up the search process. To investigate the inner structure of strings, Altunok implemented suffix trees, as described in works of Adjerho et al. [2] and Gusfield [102].

In the context of prediction algorithms based on interaction sequences represented as strings, various studies have already been conducted. Künze [153] gives an overview of the use of action prediction in adaptive user interfaces. Here, algorithms that predict the next action on the basis of a specific number of prior actions are introduced in the context of probabilistic models, such as Markov models. An example of an algorithm based on Markov models is the IPAM algorithm, developed by Davison and Hirsh [54]. A further approach is using recurring patterns to define probabilities for the recurrence of a given pattern. The PPM algorithm, developed by Cleary and Witten [44], is one example of this approach; it was extended in 1990 by Moffat [172] and in 1995 by Cleary et al. [45] to make PPM independent of a maximum Markov order.

The works of Borisov [29] and Yu [303] go a step further. Yu developed a formal modeling approach for human errors in interaction that is based on finite automatons (cf. McNaugthon [168]) with an extension of timing introduced by Alur [4]. The main goal here is to generate error automaton from a given expert model. By applying automaton to interaction sequences, certain errors can be identified during runtime, as well as offline. The information derived from this formal error detection can be used to trigger reconfiguration, to identify correct patterns for rule generation, and to instantiate these patterns based on the context in which the error occurred.

Borisov worked on extending the UIEditor framework with an analysis module that includes the approaches of Yu and Altunok. Furthermore, he started to develop and implement a software component for analyzing log files chronicling interaction with multiple users. The data structure developed for this approach will be further used for analysis based on data-mining implementations like JDMP, which was developed by Arndt [8], paired with an implementation for processing matrices, called UJMP Arndt et al. [9]; these are extended by data-mining algorithms implemented in the WEKA project, which began in 1993 by Cunningham et al. [53]. Thus, Borisov's implementation can be seen as a seminal implementation of the *Interaction Analyzer* indicated in the above described architecture for automatic reconfiguration and thus as a foundation for future work on implementing the entire concept and creation process for automatic reconfiguration.

8.4. Conclusion

This chapter anticipates the next one, which will introduce a major field for future work following this dissertation. The foundation for this work has already been laid by various students who investigated a variety of aspects of analysis of interaction sequences. Besides formal analysis of interaction logic, the analysis of interaction sequences has been investigated in three theses: Altunok investigated the role of string matching and prediction algorithms in logging-based interaction analysis and implemented a broad set of different algorithms; Borisov extended Altunok's work by focusing on multi-user scenarios and the integration of these interaction analysis tools into the UIEditor framework; Yu worked on formalizing and implementing models representing human errors in interaction, based on timed finite automatons. The next chapter will provide an overall conclusion for this dissertation and a broader perspective on future work, including the initial steps in the direction of automatic reconfiguration of user interfaces.

9. Conclusion and Future Work

The main goal of this dissertation has been to develop a formal approach to user interface modeling while differentiating between the physical representation and the interaction logic of the user interface. Adaptive interfaces provide high usability while decreasing errors in interaction. This approach led to the development of a formal language offering various modeling techniques to describe data processing between the physical representation of a user interface and the system to be controlled. This language was inspired by mental models developed in cognitive psychology. The formal reconfiguration approach also draws on work in graph rewriting applied to higher Petri nets. This entailed the development of a Java-based framework for support tools to model, simulate, and interactively reconfigure a formal user interface. In the end, this dissertation builds the foundation for further introduction of formal models into the modeling of user interfaces and to close the gap between modeling and implementation.

In the course of this work, three evaluation studies were conducted at the university of Duisburg-Essen. The first was carried out in 2009; its goal was to evaluate the influence of user interface reconfiguration on learning complex algorithms within the context of a cooperative scenario. The results showed that students using the new techniques for reconfiguration performed significantly better than the control group. The results also confirmed that mental models can be transfered to the interaction logic of a user interface.

Two studies conducted in 2010 investigated the reduction of human errors in interaction, one through the reconfiguration of the input and the other through the reconfiguration of the output parts of a formally modeled user interface. In both studies, the positive impact of reconfiguration on the number of errors made was significant. These results provided motivation for extending the approach through the use of intelligent algorithms and methods to apply fully automatic reconfiguration to any given user interface. To this end, various approaches were described and developed in three thesis projects supervised by the author of this dissertation. Thus, this promising approach to formal modeling of user interfaces has been confirmed through a series of successful empirical studies.

The formal concepts developed in this dissertation and its associated implementations open a broad area of future work in various fields and from various perspectives. Thus, the main objective of future work will be to extend this approach to a technology implementing concepts and tools that combine cognitive psychology, engineering, and HCI from the perspective of computer science. Using formal description languages, this approach can implement transformations and applications from various fields without losing information in the process of applying these concepts to computer-based systems.

Figure 9.1 provides a conceptual model of future work based on the outcomes of this dissertation. The following list describes these subjects in greater detail:

Application Scenario Nowadays, HCI is not restricted to one-user, one-interface scenarios. More often, complex interaction scenarios take place in many situations, such as the control of nuclear power plants, which is a classic example of teamwork involving the use of a complex,

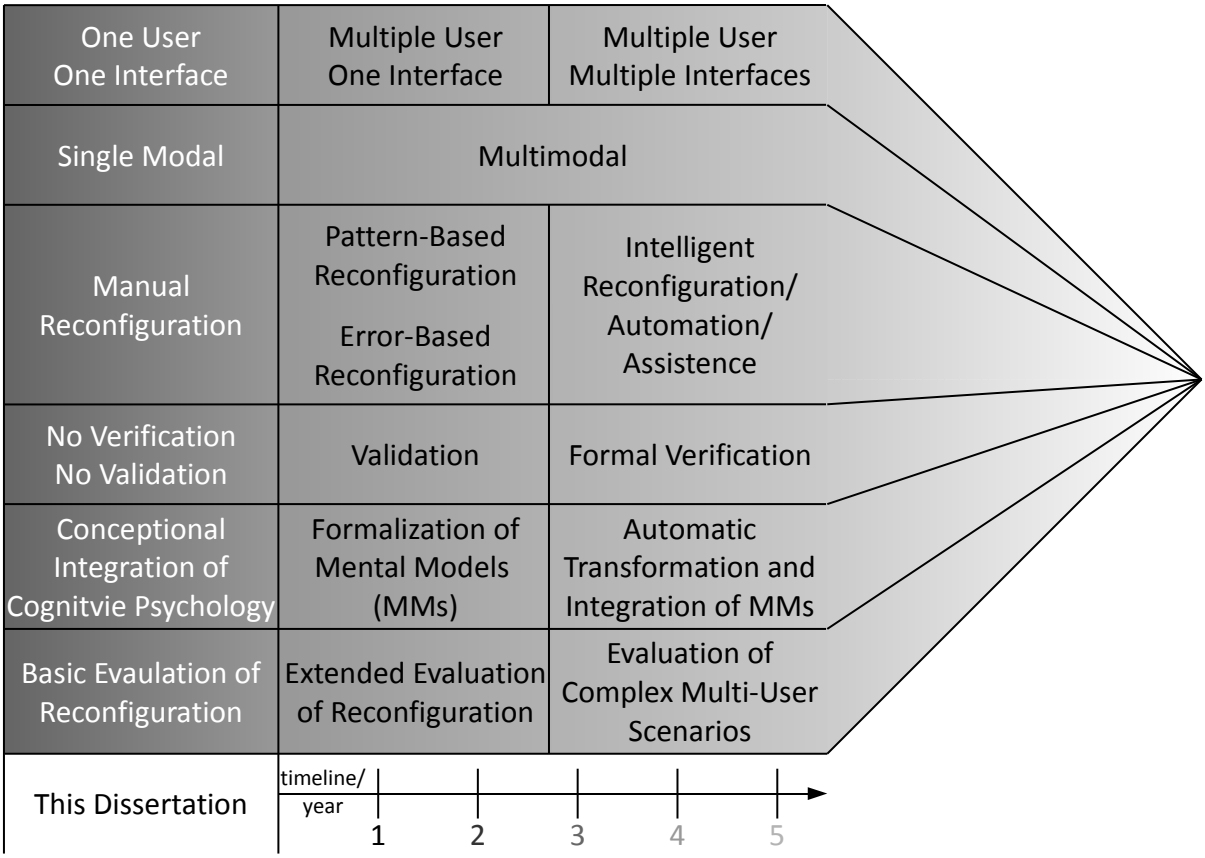


Figure 9.1.: Future work building bridges between formal modeling, cognitive psychology, and engineering

multiple-user interface. Furthermore, it is often the case that not only one specific interface is used to solve a given task; instead, various types of user interfaces are needed and are operated on various types of devices. In some cases, one interface is used by multiple users with differing purposes and issues. Thus, the first step is to identify and classify possible scenarios before solutions can be developed based on formal approaches. Cognitive psychology concepts regarding teamwork should be taken into account, paired with research on technical systems that involve control by more than one user.

Modality This dissertation concentrates on haptic input and visual output, leaving out all other possible modalities for interaction, like speech, acoustics, gestures, emotions, and so forth. Nevertheless, many studies have shown the high impact of modalities on HCI (cf. Section 2.8). Thus, modeling multi-modality must be a major aspect of future work. The approach introduced here must be broadened with regard not only to modeling but also to reconfiguration, where there is currently no formal support for multimodal user interfaces.

Reconfiguration The studies described in this dissertation dealt with user-applied reconfiguration. Future work in the area of reconfiguration was discussed in detail in Chapter 8. In addition to introducing results from interaction analysis, automation technology concepts must also be included. Here, questions concerning interaction processes and how an interface is used

to control a system in any given situation need to be answered in order to develop concepts for the application of reconfiguration to user interfaces. Then, controlling structures can be introduced into the user interface and applied by reconfiguration; also appropriate concepts for automating control can be applied to user interface models. Automation in this sense can also implement structures in the formal user interface to help the user.

Verification and Validation This thesis provides a concrete basis for future work in this area. Reference nets, as a special type of Petri nets, were chosen because of their broad range of theories and tools for validation and verification. Especially in process modeling, various studies have been conducted that can be applied to the formal modeling of interaction logic in order to answer questions concerning such issues as accessibility. The introduction of verification and validation concepts can be applied not only to a modeled user interface but also to its reconfiguration. This enables mental models to be built and described and automation concepts to be identified and verified. For instance, the correctness of reconfigured user interfaces can be investigated before the user interface is changed.

Integration of Cognitive Psychology In this dissertation, results from psychology informed the overall structure of interaction logic models. Still, a formal integration of cognitive psychology was not developed and described. Therefore, an important objective for future work is the closer integration of concepts and modeling approaches from cognitive psychology. Thus, formalization of mental models and learning them from interaction are only two aspects of a broad set of issues concerning the formal integration of cognitive psychology into formal and adaptive user interfaces. Moreover, various modeling approaches from engineering can be introduced directly into the modeled user interface depending on their formal basis.

Evaluation Certain aspects of the other concepts and tools introduced in the course of these studies remain to be evaluated—namely those related to their performance in enhancing interaction between human users and complex technical systems in extended scenarios and additional technical modules.

Other aspects are also of interest. The first is new application scenarios, like virtual museums and smart-city environments. Next, the application of the formal modeling approach to smart living environments paired with an inversion of the roles of user and user interface plays a central role in future work. Here, the system perceives the user's behavior and his activities, using those perceptions to provide relevant tools and services to the user via the interface. In that sense, the system may be seen as controlling the user through the adaption of the user interface. Another aspect is the involvement of models of the system in the reconfiguration; here, changes applied to the interaction logic can be embedded in the system model and vice versa. Thus, the system can learn from the user's interaction and reconfiguration of the user interface, and the interaction logic can be specified by the system if critical system states can be reached.

In conclusion, in this dissertation a formal approach to user interface modeling was developed that opens the door to a common basis for various disciplines interested in HCI. The use of the implemented framework evinced a positive impact on HCI in various evaluation studies with different objectives. This work suggested a broad range of topics for future work. A discussion of those topics concluded this dissertation, revealing its many potential extensions.

A. The UIEditor Framework—File Formats

In this appendix, file formats used in the UIEditor framework will be specified using RelaxNG Schema for defining XML-based file formats. First, the `.uie` file format will be introduced followed by the `.sim` format. In a further section, the extended PNML format developed by Stückrath [265] for serialization of reference nets will be introduced accompanied with the specification of the format for defining DPO rules for formal reconfiguration. The UIEditor framework uses the Java-based open source library Jing[1] for file validation.

UIEditor Framework File Formats

`.uie` File Format

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <grammar xmlns="http://relaxng.org/ns/structure/1.0">
4
5         <start>
6                 <element name="uiconfig">
7                         <ref name="interfaces"/>
8                         <ref name="physrep"/>
9                         <ref name="interactionloc"/>
10                </element>
11        </start>
12
13        <define name="interfaces">
14                <element name="programminterfaces">
15                        <interleave>
16                                <oneOrMore>
17                                        <element name="systeminterface">
18                                                <attribute name="path"/>
19                                        </element>
20                                </oneOrMore>
21                                <zeroOrMore>
22                                        <element name="ilopinterface">
23                                                <attribute name="path"/>
24                                        </element>
25                                </zeroOrMore>
26                        </interleave>
27                </element>
28        </define>
29
30        <define name="physrep">
31                <element name="design">
32                        <zeroOrMore>
33                                <ref name="physrep.ielements"/>
```

[1] http://code.google.com/p/jing-trang

```
34                      </zeroOrMore>
35              </element>
36      </define>
37
38      <define name="physrep.ielements">
39              <element name="widget">
40                      <attribute name="id"/>
41                      <attribute name="type"/>
42                      <attribute name="name"/>
43                      <interleave>
44                              <ref name="element.bounds"/>
45                              <zeroOrMore>
46                                      <element name="parameter">
47                                              <attribute name="name"/>
48                                              <attribute name="datatype"/>
49                                              <attribute name="value"/>
50                                      </element>
51                              </zeroOrMore>
52                              <element name="interface">
53                                      <ref name="physrep.ielements.interface"/>
54                              </element>
55                      </interleave>
56              </element>
57      </define>
58
59      <define name="physrep.ielements.interface">
60              <zeroOrMore>
61                      <element name="widgetport">
62                              <attribute name="id"/>
63                              <attribute name="name"/>
64                              <attribute name="datatype"/>
65                              <attribute name="index"/>
66                              <ref name="element.function"/>
67                      </element>
68              </zeroOrMore>
69              <zeroOrMore>
70                      <element name="event">
71                              <attribute name="id"/>
72                              <attribute name="type"/>
73                              <attribute name="datatype"/>
74                              <attribute name="value"/>
75                              <ref name="element.function"/>
76                      </element>
77              </zeroOrMore>
78      </define>
79
80      <define name="interactionloc">
81              <zeroOrMore>
82                      <element name="ilmodule">
83                              <interleave>
84                                      <element name="operations">
85                                              <ref name="interactionloc.ops"/>
86                                      </element>
87                                      <element name="connectors">
88                                              <ref name="interactionloc.con"/>
89                                      </element>
```

```
90                              </interleave>
91                      </element>
92              </zeroOrMore>
93      </define>
94
95      <define name="interactionloc.ops">
96              <zeroOrMore>
97                      <element name="bpmn">
98                              <attribute name="id"/>
99                              <attribute name="type"/>
100                             <attribute name="inscription"/>
101                             <ref name="element.bounds"/>
102                     </element>
103             </zeroOrMore>
104             <zeroOrMore>
105                     <element name="channeloperation">
106                             <attribute name="id"/>
107                             <attribute name="type"/>
108                             <attribute name="name"/>
109                             <interleave>
110                                     <element name="channel">
111                                             <attribute name="id"/>
112                                             <attribute name="name"/>
113                                     </element>
114                                     <ref name="interactionloc.ops.param"/>
115                             </interleave>
116                     </element>
117             </zeroOrMore>
118             <zeroOrMore>
119                     <element name="iloperation">
120                             <interleave>
121                                     <zeroOrMore>
122                                             <element name="initialparameter">
123                                               <attribute name="name"/>
124                                               <attribute name="datatype"/>
125                                               <attribute name="value"/>
126                                             </element>
127                                     </zeroOrMore>
128                                     <ref name="interactionloc.ops.param"/>
129                             </interleave>
130                     </element>
131             </zeroOrMore>
132             <zeroOrMore>
133                     <element name="systemoperation">
134                             <ref name="interactionlog.ops.param"/>
135                     </element>
136             </zeroOrMore>
137     </define>
138
139     <define name="interactionloc.ops.param">
140             <interleave>
141                     <ref name="element.function"/>
142                     <ref name="element.bounds"/>
143                     <optional>
144                             <element name="description"><text/></element>
145                     </optional>
```

```
146                         <zeroOrMore>
147                                 <element name="iport">
148                                         <attribute name="id"/>
149                                         <attribute name="datatype"/>
150                                 </element>
151                         </zeroOrMore>
152                         <zeroOrMore>
153                                 <element name="oport">
154                                         <attribute name="id"/>
155                                         <attribute name="datatype"/>
156                                 </element>
157                         </zeroOrMore>
158                 </interleave>
159         </define>
160
161         <define name="interactionloc.con">
162                 <zeroOrMore>
163                         <element name="connector">
164                                 <attribute name="source"/>
165                                 <attribute name="target"/>
166                                 <attribute name="type"/>
167                                 <attribute name="inscription"/>
168                         </element>
169                 </zeroOrMore>
170         </define>
171
172         <define name="element.bounds">
173                 <element name="bounds">
174                         <attribute name="x"><text/></attribute>
175                         <attribute name="y"><text/></attribute>
176                         <attribute name="width"><text/></attribute>
177                         <attribute name="height"><text/></attribute>
178                 </element>
179         </define>
180
181         <define name="element.function">
182                 <element name="function">
183                         <attribute name="name"/>
184                 </element>
185         </define>
186
187 </grammar>
```

`.sim` File Format

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <grammar xmlns="http://relaxng.org/ns/structure/1.0">
4
5         <start>
6                 <element name="simulation">
7                         <interleave>
8                                 <element name="pnml">
9                                         <attribute name="path"/>
10                                        <attribute name="netname"/>
11                                </element>
12                                <element name="stub">
```

```
13                                      <attribute name="path"/>
14                              </element>
15                              <element name="ui">
16                                      <attribute name="path"/>
17                              </element>
18                              <zeroOrMore>
19                                      <element name="systeminterface">
20                                              <attribute name="path"/>
21                                      </element>
22                              </zeroOrMore>
23                              <zeroOrMore>
24                                      <element name="iloperationinterface">
25                                              <attribute name="path"/>
26                                      </element>
27                              </zeroOrMore>
28                      </interleave>
29              </element>
30      </start>
31
32 </grammar>
```

Extension of PNML to Colored Petri Nets

The extension of PNML format introduced below uses the overriding concept of RelaxNG similar to concepts in object-orientation.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3       <grammar xmlns="http://relaxng.org/ns/structure/1.0">
4
5       <include href="basicPNML.rng">
6               <define name="nettype.uri" combine="choice">
7                       <value>refNet</value>
8               </define>
9       </include>
10
11      <define name="net.labels" combine="interleave">
12              <interleave>
13                      <optional>
14                              <element name="name">
15                                      <ref name="text.node"/>
16                              </element>
17                      </optional>
18                      <optional>
19                              <element name="declaration">
20                                      <ref name="inscription.content"/>
21                              </element>
22                      </optional>
23              </interleave>
24      </define>
25
26      <define name="arc.labels" combine="interleave">
27              <interleave>
28                      <optional><ref name="inscription.label"/></optional>
29                      <optional><ref name="arc.type"/></optional>
30              </interleave>
31      </define>
```

```
32
33          <define name="arc.type">
34                  <element name="type">
35                          <ref name="text.node"/>
36                  </element>
37          </define>
38
39          <define name="place.labels" combine="interleave">
40                  <optional><ref name="inscription.label"/></optional>
41          </define>
42
43          <define name="transition.labels" combine="interleave">
44                  <optional><ref name="inscription.label"/></optional>
45          </define>
46
47          <define name="inscription.label">
48                  <element name="inscription">
49                          <ref name="inscription.content"/>
50                  </element>
51          </define>
52
53          <define name="inscription.content">
54                  <interleave>
55                          <optional>
56                                  <element name="graphics">
57                                          <ref name="annotationgraphics.content"/>
58                                  </element>
59                          </optional>
60                          <ref name="text.node"/>
61                  </interleave>
62          </define>
63
64          <define name="text.node">
65                  <element name="text">
66                          <text/>
67                  </element>
68          </define>
69
70 </grammar>
```

Definition of DPO Rules

In a first step, Stückrath defined a generic rule schema specifying the major structure of a rule definition. In a second step, this generic definition was specified by applying the extended PNML schema for colored Petri nets.

Generic Rule Definition

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <grammar xmlns="http://relaxng.org/ns/structure/1.0"
4                   datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
5
6          <start>
```

```
7               <ref name="trans.rule"/>
8       </start>
9
10      <define name="trans.rule">
11              <element name="rule">
12                      <interleave>
13                              <ref name="trans.left"/>
14                              <ref name="trans.interface"/>
15                              <ref name="trans.right"/>
16                              <ref name="trans.mapping"/>
17                      </interleave>
18              </element>
19      </define>
20
21      <define name="trans.left">
22              <element name="deleteNet">
23                      <ref name="net.definition"/>
24              </element>
25      </define>
26
27      <define name="trans.interface">
28              <element name="interface">
29                      <ref name="net.definition"/>
30              </element>
31      </define>
32
33      <define name="trans.right">
34              <element name="insertNet">
35                      <ref name="net.definition"/>
36              </element>
37      </define>
38
39      <define name="net.definition">
40              <empty/>
41      </define>
42
43      <define name="trans.mapping">
44              <element name="mapping">
45                      <zeroOrMore>
46                              <ref name="trans.mapping.element"/>
47                      </zeroOrMore>
48              </element>
49      </define>
50
51      <define name="trans.mapping.element">
52              <element name="mapElement">
53                      <attribute name="interfaceID">
54                              <data type="ID"/>
55                      </attribute>
56                      <attribute name="deleteID">
57                              <data type="ID"/>
58                      </attribute>
59                      <attribute name="insertID">
60                              <data type="ID"/>
61                      </attribute>
62              </element>
```

```
63          </define>
64
65 </grammar>
```

Reference Net Rule Definition

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <grammar xmlns="http://relaxng.org/ns/structure/1.0"
4                   datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
5
6          <include href="refPNML.rng">
7                  <start combine="interleave">
8                          <empty/>
9                  </start>
10         </include>
11
12         <include href="generic_rule.rng"/>
13
14         <define name="net.definition" combine="interleave">
15                 <ref name="net.element"/>
16         </define>
17
18 </grammar>
```

B. The UIEditor Framework—Availability

Availability

It is planned to provide the UIEditor framework as open source project after finishing the future work implementation introduced in Chapter 8 and 9. Table B.1 shows all used libraries in the UIEditor framework as preparation for publicize its source code.

Table B.1.: Used libraries in the UIEditor framework implementation

Library	License	URL
JFreeChart	GNU Lesser Public License	`http://www.jfree.org/jfreechart/`
JCommon	GNU Lesser Public License	`http://www.jfree.org/jcommon/`
JGoodies	BSD Open Source	`http://www.jgoodies.com/downloads/`
JDOM	Apache Open Source	`http://www.jdom.org/`
Renew	GNU Lesser Public License	`http://www.renew.de`
Log4j	Apache License, version 2	`http://logging.apache.org/log4j/1.2/`
JGraph	BSD Open Source	`http://www.jgraph.com/jgraph.html`
Jing	Thai Open Source Software Center Ltd.	`http://www.thaiopensource.com/relaxng/jing.html`

List of Figures

List of Tables

Bibliography

[1] J. Adamek, H. Herrlich, and G. E. Stecker. *Abstract and Concrete Categories*. John Wiley and Sons, 1999.

[2] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2010.

[3] C. Altunok. Serialisierung von Interaktionsverläufen und automatisches Erkennen von Interaktionsmustern. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2011.

[4] R. Alur. Timed automata. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 688–689. Springer, 1999.

[5] J. R. Anderson. *Cognitive Psychology and Its Implications*. W. H. Freeman & Co Ltd., 5th edition, 2000.

[6] J. R. Anderson, C. F. Boyle, R. Farrell, and B. J. Reiser. Cognitive principles in the design of computer tutors. In P. Morris, editor, *Modelling Cognition*, chapter 4. John Wiley & Sons Ltd., 1987.

[7] J. H. Andrews. *Logic Programming*. Cambridge University Press, 1992.

[8] H. Arndt. The Java data mining package—A data processing library for Java. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, COMPSAC'09, pages 620–621, Seattle, WA, USA, 2009.

[9] H. Arndt, M. Bundschus, and A. Nägele. Towards a next-generation matrix library for Java. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, COMPSAC'09, pages 460–467, Seattle, WA, USA, 2009.

[10] S. Awodey. *Category Theory*. Oxford University Press, 2010.

[11] M. Azmitia. Peer interaction and problem solving: When are two heads better than one? *Child Development*, 59(1):87–96, 1988.

[12] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, Volume I*, pages 447–533. Elsevier Science, 2001.

[13] J. Bagga and A. Heinz. JGraph—A Java based system for drawing graphs and running graph algorithms. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 459–460. Springer, 2002.

[14] N. Baloian, H. Breuer, and W. Luther. Concept keyboards in the animation of standard algorithms. *Journal of Visual Languages and Computing*, 19(6):652–674, 2008.

[15] N. Baloian, H. Breuer, W. Luther, and C. Middelton. Algorithm visualization using concept keyboards. In *Proceedings of ACM Conference on Software Visualization*, SoftVis'05, pages 7–16, St. Louis, MO, USA, 2005.

[16] N. Baloian and W. Luther. Modeling educational software for people with disabilities: Theory and practice. In *Proceedings of International Conference on Computers and Accessibility*, ASSETS'02, pages 111–118, Edinburgh, Scotland, UK, 2002.

[17] N. Baloian and W. Luther. Algorithm explanation using multimodal interfaces. In *Proceedings of the 25th International Conference of the Chilean Computer Science Society*, page 21, Washington, DC, USA, 2005.

[18] N. Baloian and W. Luther. Cooperative visualization of cryptographic protocols using concept keyboards. *International Journal of Engineering and Education*, 25(4):745–754, 2009.

[19] R. Bastide and P. A. Palanque. Petri net objects for the design, validation and prototyping of user-driven interfaces. In *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction*, INTERACT '90, pages 625–631, Cambridge, UK, 1990.

[20] R. Bastide and P. A. Palanque. A visual and formal glue between application and interaction. *Journal of Visual Languages and Computing*, 10(5):481–507, 1999.

[21] B. Baumgarten. *Petri Netze. Grundlagen und Anwendungen.* Spektrum Akademischer Verlag, 2nd edition, 1996.

[22] F. Bause and P. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory.* Vieweg Verlag, 1996.

[23] R. Bentley, T. Rodden, P. Sawyer, and I. Sommerville. An architecture for tailoring cooperative multi-user displays. In *Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work*, CSCW '92, pages 187–194, Toronto, Ontario, Canada, 1992.

[24] BPMN Offensive Berlin. BPMN 2.0—Business Process Model and Notation, online, URL: `http://bpmn.de/poster`, last visited: 10-11-2011.

[25] L. Bernardinello and F. De Cindio. A survey of basic net models and modular net classes. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 609 of *Lecture Notes in Computer Science*, pages 304–351. Springer, 1992.

[26] N. Bertrand, G. Delzanno, B. König, A. Sangier, and J. Stückrath. On the decidability status of reachability and coverability in graph transformation systems. In *Proceedings of International Conference on Rewriting Techniques and Applications*, RTA '12, submitted, 2012.

[27] E. Best and C. Fernandez. *Notations and Terminology on Petri Net Theory*, volume 185 of *Working Papers.* GMD, 1987.

[28] J. Borchers. *A pattern approach to interaction design.* John Wiley, 2001.

[29] N. Borisov. Kontextabhängige Modellierung und Visualisierung von Interaktionsverläufen in Mehrbenutzersystemen—Konzept und Realisierung. Master's thesis, University of Duisburg-Essen, Departement of Computer Science and Applied Cognitive Science, Germany, 2012.

[30] M. C. Bos. Experimental study of productive collaboration. *Acta Psychologica*, 3:315–426, 1937.

[31] D. A. Bowman, E. Kruijff, and J. J. Laviola. *3D User Interfaces: Theory and Practice*. Addison-Wesley Longman, 2004.

[32] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of ACM*, 20(10):762–772, 1977.

[33] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report: Structural layer proposal. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 501–512. Springer, 2002.

[34] M. H. Brown. Perspectives on algorithm animation. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '88, pages 33–38, Washington, DC, USA, 1988.

[35] G. Calvary, J. Coutaz, and L. Nigay. From single-user architectural design to PAC*: A generic software architecture model for CSCW. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '97, pages 242–249, Atlanta, GA, USA, 1997.

[36] A. Cansado, C. Canal, G. Salaün, and J. Cubo. A formal framework for structural reconfiguration of components under behavioural adaptation. *Electronic Notes in Theoretical Computer Science*, 263:95–110, 2010.

[37] X. Cao, C. Forlines, and R. Balakrishnan. Multi-user interaction using handheld projectors. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 43–52, New Port, RI, USA, 2007.

[38] S. K. Card, T. P. Moran, and A. Newell. Computer text-editing: An information-processing analysis of a routine cognitive skill. *Cognitive Psychology*, 12(1):32–74, 1980.

[39] S. K. Card, T. P. Moran, and A. Newell. The keystroke-level model for user performance time with interactive systems. *Communications of ACM*, 23(7):396–410, 1980.

[40] S. K. Card, T. P. Moran, and A. Newell. *The psychology of human-computer interaction*. CRC Press, 2008.

[41] J. Carroll and J. Olson. Mental models in human-computer interaction. In J. A. Jacko and A. Sears, editors, *Handbook of Human-Computer Interaction*, pages 45–65. Elsevier, 1988.

[42] J. L. Casti. *Nonlinear System Theory*, volume 175 of *Mathematics in Science and Engineering*. Academic Press, 1985.

[43] S. Christensen and N. Damgaard Hansen. Coloured petri nets extended with channels for synchronous communication. In R. Valette, editor, *Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 1994.

[44] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions of Communications*, 32(4):396–402, 1984.

[45] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. *The Computer Journal*, 40(2–3):67–75, 1997.

[46] V. Colella. Participatory simulations: Building collaborative understanding through immersive dynamic modeling. *The Journal of the Learning Sciences*, 9(4):471–500, 2000.

[47] IMS Global Learning Consortium. IMS Learner Information Package Specification, online, URL: `http://www.imsglobal.org/profiles/`, last visited: 10-11-2011.

[48] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation—Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of graph grammars and computing by graph transformations*, pages 163–246. World Scientific, 1997.

[49] G. Coulouris, J. Dollimore, and R. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley Longman, 4th edition, 2005.

[50] J. Coutaz. PAC, an object oriented model for dialog design. In *Poceedings of International Conference on Human-Comuter Interaction*, Interact '87, pages 431–436, Stuttgart, Germany, 1987.

[51] D. Crane, E. Pascarello, and D. James. *Ajax in Action.* Manning Publications, 2005.

[52] P. Crescenzi, N. Faltin, R. Fleischer, C. Hundhausen, S. Näher, G. Rössling, J. Stasko, and E. Sutinen. The algorithm animation repository. In *Proceedings of the Second International Program Visualization Workshop*, pages 14–16, Arhus, Denmark, 2002.

[53] S. J. Cunningham and P. Denize. A tool for model generation and knowledge acquisition. In *Proceedings of International Workshop on Artificial Intelligence and Statistics*, AISTATS '93, pages 213–222, Fort Lauderdale, FL, USA, 1993.

[54] D. B. Davison and H. Hirsh. Predicting sequences of user actions. In *Workshop on Predicting the Future: AI Approaches to Time-Series Problems*, ICML '98, pages 5–12, Madison, WI, USA, 1998.

[55] F. de Rosi, S. Pizzutilo, and B. de Carolis. Formal description and evaluation of user-adapted interfaces. *International Journal of Human-Computer Studies*, 49(2):95–120, 1998.

[56] S. W. Dekker. *Ten Questions about Human Error: A New View of Human Factors and System Safety.* Lawrence Erlbaum, 2005.

[57] W. E. Deming. *Out of the Crisis.* McGraw-Hill Inc., 1986.

[58] J. Dieckmann. *Einführung in die Systemtheorie*, volume 8305 of *UTB für Wissenschaft.* W. Fink Verlag, 2005.

[59] S. Diehl. Future perspective—Introduction. In S. Diehl, editor, *Software Visualization—State-of-the-Art Survey*, volume 2269 of *Lecture Notes in Computer Science*. Springer, 2002.

[60] S. Diehl. Software visualization. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 718–719, St. Louis, MO, USA, 2005.

[61] S. Diehl and T. Kunze. Visualizing principles of abstract machines by generating interactive animations. *International Journal of Future Generation Computer Systems*, 16(7):831–839, 2000.

[62] H. Dietrich, U. Malinowski, T. Kühne, and M. Schneider-Hufschmidt. State of the art in adaptive user interfaces. In M. Schneider-Hufschmidt, T. Kühme, and U. Malinowski, editors, *Adaptive User Interfaces: Principles and Practice*, pages 13–48. Elsevier, 1993.

[63] R. Dini, F. Paternò, and C. Santoro. An environment to support multi-user interaction and cooperation for improving museum visits through games. In *Proceedings of the 9th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '07, pages 515–521, Singapore, 2007.

[64] A. Dix, J. Finlay, G. D. Abowd, and R. Beale. *Human-computer interaction*. Pearson Prentice Hall, 3rd edition, 2004.

[65] K. C. Dohse, T. Dohse, J. D. Still, and D. J. Parkhurst. Enhancing multi-user interaction with multi-touch tabletop displays using hand tracking. In *Proceedings of the 1st International Conference on Advances in Computer-Human Interaction*, ACHI '08, pages 297–302, Saint Luce, France, 2008.

[66] Y. Donchin, D. Gophe, M. Olin, Y. Badihi, M. Biesky, C. L. Sprung, R. Pizov, and S. Cotev. A look into the nature and causes of human errors in the intensive care unit. *Critical Care Medicin*, 23(2):294–300, 1995.

[67] D. Dörner. *Logik des Misslingens*. Rowohlt Verlag, 10th edition, 2003.

[68] G. Douglas and M. McLinden. The use of concept keyboards to teach early tactile reading. *Eye Contact*, 19:31–33, 1997.

[69] E. Edwards. Introductory overview. In E. Wiener and D. Nagel, editors, *Human Factors in Aviation*, pages 3–25. Academic, 1988.

[70] H. Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Proceedings of International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, pages 1–69, Bad Honnef, Germany, 1978.

[71] H. Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In *Proccedings of the 3rd International Workshop of Graph Grammars and their Application to Computer Science*, volume 291 of *Lecture Notes of Computer Science*, pages 3–14. Springer, 1986.

[72] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999.

[73] 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 G. Rozenberg, editor, *Handbook of graph grammars and computing by graph transformation*, chapter 4. World Scientific Publishing, 1997.

[74] H. Ehrig, K. Hoffmann, and J. Padberg. Transformation of Petri nets. *Electronic Notes in Theoretical Computer Science*, 148(1):151–172, 2006.

[75] H. Ehrig, K. Hoffmann, J. Padberg, C. Ermel, U. Prange, E. Biermann, and T. Modica. Petri net transformation. In V. Kordic, editor, *Petri Net, Theory and Applications*, chapter 1. InTech Education and Publishing, 2008.

[76] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[77] H. Ehrig and J. Padberg. Graph grammars and Petri net transformations. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes of Computer Science*, pages 65–86. Springer, 2004.

[78] M. Eisenberg. The thin glass line: Designing interfaces to algorithms. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems: Common Ground*, CHI '96, pages 181–188, Reading, MA, USA, 1996.

[79] J. Eisenstein and A. Puerta. Adaptation in automated user-interface design. In *Proceedings of the 5th International Conference on Intelligent User Interfaces*, IUI '00, pages 74–81, New Orleans, LA, USA, 2000. ACM.

[80] M. Elkoutbi and R. K. Keller. User interface prototyping based on UML scenarios and high-level petri nets. In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, ICATPN '00, pages 166–186, Aarhus, Denmark, 2000.

[81] A. Emeljanov. Charakterisierung von Ausgabeelementen zur Rekonstruktion von Benutzerschnittstellen—Formale Beschreibung und Implementierung. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2010.

[82] M. R. Endsley. Measurement of situation awareness in dynamic systems. *Human Factors*, 37(1):65–84, 1995.

[83] M. R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors*, 37(1):32–64, 1995.

[84] M. R. Endsley. Errors in situation assessment: Implications for system design. In P. F. Elzer, R. H. Kluwe, and B. Boussoffara, editors, *Human Error and System Design and Management*, pages 15–26. Springer, 2000.

[85] A. Ericsson and W. Kintsch. Long-term working memory. *Psychological Review*, 102(2):211–245, 1995.

[86] H. Eriksson. Control The Nuclear Power Plant, online, URL: `http://www.ida.liu.se/~her/npp/demo.html`, last visited: 10-14-2011.

[87] M. Eysenck. *Fundamentals of Cognition.* Psychology Press, 2011.

[88] C. Faulkner. *The Essence of Human Computer Interaction.* Prentice Hall, 2001.

[89] K. C. Feldt. *Programming Firefox: Building Applications in the Browser.* O'Reilly, 2007.

[90] A. Field. *Discovering Statistics Using SPSS (Introducing Statistical Method).* Sage Publications Ltd., 3rd edition, 2009.

[91] G. Fischer. User modeling in human computer interaction. *User Modeling and User-Adapted Interaction*, 11(1):65–86, 2001.

[92] D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly, 2011.

[93] X. Fu, D. Gamrad, H. Mosebach, K. Lemmer, and D. Söffker. Modeling and implementation of cognitive-based supervision and assistance. In *Proceedings of 6th Vienna Conference on Mathematical Modeling on Dynamical Systems*, MATHMOD '09, pages 2063–2068, Vienna, Austria, 2009.

[94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, 2nd edition, 1994.

[95] D. Gamrad and D. Söffker. Implementation of a novel approach for the simulation of cognition based on situation-operator-modeling and high-level petri nets. In *Proceedings of ICROS-SICE International Joint Conference*, ICCAS-SICE '09, pages 1404–1410, Fukuoka, Japan, 2009.

[96] D. Gamrad and D. Söffker. Simulation of learning and planning by a novel architecture for cognitive technical systems. In *Proceedings of IEEE Interantional Conference on System, Man, and Cybernetics*, SMC '09, pages 2302–2307, San Antonio, TX, USA, 2009.

[97] F. Gerantabee and AGI Creative Team. *Flash Professional CS5 Digital Classroom.* Wiley, 2010.

[98] C. Girault and R.Valk. *Petri Nets for System Engineering.* Springer, 2003.

[99] A. Goldberg. *SMALLTALK-80: The interactive programming environment.* Addison-Wesley, 1984.

[100] T. Gross and M. Koch. *Computer-Supported Cooperative Work.* Oldenbourg Verlag, 2007.

[101] H.-P. Gumm, M. Sommer, W. Hesse, and B. Seeger. *Einführung in die Informatik.* Oldenbourg Verlag, 2001.

[102] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[103] P. A. Hancock. Individuation: The n = 1 revolution. *Theoretical Issues in Ergonomics Science*, 10(5):481–488, 2009.

[104] P. A. Hancock, A. A. Pepe, and L. L. Murphy. Hedonomics: The power of positive and pleasurable ergonomics. *Ergonomics in Design*, 13(1):8–14, 2005.

[105] C. Hand. A survey of 3d interaction techniques. *Computer Graphics Forum*, 16(5):269–281, 1997.

[106] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[107] R. Heckel. Introductory tutorial on foundations and applications of graph transformation. In A. Corradini, H. Ehrig, U. Montanari, L. Riberio, and G. Rozenberg, editors, *Graph Transformations*, volume 4178 of *Lecture Notes of Computer Science*, pages 461–462. Springer, 2006.

[108] C. Heij, A. C. Ran, and F. van Schagen. *Introduction to Mathematical Systems Theory: Linear Systems, Identification and Control*. Birkhäuser Basel, 2006.

[109] I. Hermann and M. S. Marshall. GraphXML—An XML-based graph desription format. In *Proceedings of Symposium on Graph Drawing*, GD '00, Colonial Williamsburg, VA, USA, 2000.

[110] 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*, GCM '10, Enschede, The Netherlands, 2010.

[111] R. D. Hill. The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '92, pages 335–342, Monterey, CA, USA, 1992.

[112] L. M. Hillah, E. Kindler, F. Kordon, L. Pertrucci, and N. Trèves. A primer on the Petri net markup language and ISO/IEC 15909-2. *Petri Net Newsletter*, 76:9–28, 2009.

[113] E. Hollnagel. *Cognitive Reliability and Error Analysis Method*. Elsevier, 1998.

[114] E. Hollnagel, M. Kaarstad, and H.-C. Lee. Error mode prediction. *Ergonomics*, 24(11):1457–1471, 1999.

[115] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, Addison Wesley, 3rd edition, 2010.

[116] C. D. Hundhausen. *Toward effective algorithm visualization artifacts: Designing for participation and communication in an undergraduate algorithms course*. PhD thesis, University of Hawaii at Manoa, USA, 1999.

[117] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. *Human Computer Interaction*, 1(4):311–338, 1985.

[118] ECMA International. ECMAScript language specification—ECMA-262, online, URL: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, last visited: 10-11-2011.

[119] ISO. ISO8879 information processing—text and office systems—standard generalized markup language (SGML), online, URL: `http://www.iso.org/iso/catalogue\_detail.htm?csnumber=16387`, last visited: 10-11-2011.

[120] ISO/IEC. ISO/IEC 14977:1996(e) - final draft of standard for EBNF, online, URL: `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`, last visited: 10-11-2011.

[121] A. Jameson. Adaptive interfaces and agents. In J. A. Jacko and A. Sears, editors, *Human-Computer Interaction Handbook*, pages 305–330. Erlbaum, 2003.

[122] 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*, CHI '93, pages 418–423, New York, NY, USA, 1993.

[123] M. Jeckle. Unified Modeling Language (UML), online, URL: `http://www.jeckle.de/unified.htm`, last visited: 10-11-2011.

[124] F. V. Jensen. *Bayesian Networks and Decision Graphs*. Springer, 2001.

[125] K. Jensen. *Coloured Petri Nets—Analysis Methods*, volume 2 of *Monographs on Theoretical Computer Science*. Springer, 2nd edition, 1994.

[126] K. Jensen. *Coloured Petri Nets—Basic Concepts*, volume 1 of *Monographs on Theoretical Computer Science*. Springer, 2nd edition, 1997.

[127] K. Jensen. *Coloured Petri Nets—Practical Use*, volume 3 of *Monographs on Theoretical Computer Science*. Springer, 1997.

[128] K. Jensen, L. Kristensen, and L. Wells. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, 2007.

[129] M. Jüngel, E. Kindler, and M. Weber. Towards a generic interchange format for Petri nets—Position paper. In *Proceedings of 21st Meeting on XML/SGML based Interchange Formats for Petri Nets*, ICATPN '00, pages 1–5, Aarhus, Denmark, 2000.

[130] E. Kaiser, A. Olwal, D. McGee, H. Benko, A. Corradini, X. Li, P. Cohen, and S. ven Feiner. Mutual disambiguation of 3d multimodal interaction in augmented and virtual reality. In *Proceedings of the 5th International Conference on Multimodal Interfaces*, ICMI '03, pages 12–19, Vancouver, B.C., Canada, 2003.

[131] S. Kanat. Visueller Editor zur Erstellung komplexer Interaktionselemente unter Verwendung von XML-basierter Beschreibungssprachen. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2010.

[132] K. Katsurada, Y. Nakamura, H. Yamada, and T. Nitta. XISL: A language for describing multimodal interaction scenarios. In *Proceedings of the 5th International Conference on Multimodal Interfaces*, ICMI '03, pages 281–284, Vancouver, B.C., Canada, 2003.

[133] A. Kerren, T. Müldner, and E. Shakshuki. Novel algorithm explanation techniques for improving algorithm teaching. In *Proceedings of the 2006 ACM Symposium on Software visualization*, SOFTVIS '06, pages 175–176, Brighton, UK, 2006.

[134] D. Kieras and P. G. Polson. An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies*, 22(4):365–394, 1985.

[135] R. H. Kluwe. Informationsaufnahme und Informationsverarbeitung. In B. Zimolong and U. Konradt, editors, *Ingenierpsychologie*, chapter 2. Hogrefe, 2006.

[136] R. H. Kluwe and H. Haider. Modelle zur internen Repräsentation technischer Systeme. *Sprache und Kognition*, 4(9):173–192, 1990.

[137] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[138] H. Kohls and P. Berdys. Bewertung der Rekonfiguration von Ausgabeschnittstellenelementen hinsichtlich der Steuerung eines zu implementierenden Beispielprozesses. Master's thesis, Univerisity of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2010.

[139] P. Koopman, R. Plasmeijer, and D. Swierstra. *Advanced Functional Programming*. Springer, 2009.

[140] D. Kopec, M. H. Kabir, D. Reinharth, O. Rothschild, and J. A. Castiglione. Human errors in medical practice: Systematic classification and reduction with automated information systems. *Journal of Medical Systems*, 27(4):297–313, 2003.

[141] T. Koshman, A. Kelson, P. Feltovich, and H. Barrows. Computer supported problem-based learning: A principled approach to the use of computers in collaborative learning. In T. Koschmann, editor, *CSCL: Theory and Practice of an Emerging Paradigm*, pages 87–124. Lawrence Erlbaum, 1995.

[142] T. Kotthäuser, A. Kovácová, W. Liu, W. Luther, L. Selvanadurajan, M. Wander, S. Wang, B. Weyers, A. Yapo, and K. Zhu. Concept Keyboards zur Steuerung und Visualisierung interaktiver krypthographischer Algorithmen. Technical report, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2006.

[143] A. Kovácová. Implementierung des Needham-Schroeder Protokolls in einer verteilten Simulationsumgebung für kryptografische Standardverfahren. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2007.

[144] P. Kraft. Algorithmenvisualisierung mit selbstkonfigurierten Schnittstellen—Implementation und Evaluation. Master's thesis, Univerisity of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2005.

[145] M. Kreuzer and S. Kühling. *Logik für Informatiker*. Pearson, 2006.

[146] O. Kummer. A Petri net view on synchronous channels. *Petri Net Newsletter*, 56:7–11, 1999.

[147] O. Kummer. Tight integration of Java and Petri nets. In *Proceedings of 6th Workshop of Algorithms and Tools for Petri Nets*, AWPN '99, pages 30–35, Frankfurt, Germany, 1999.

[148] O. Kummer. Introduction to Petri nets and reference nets. *Sozionik Aktuell*, 1:1–9, 2001.

[149] O. Kummer. *Referenznetze*. PhD thesis, University of Hamburg, Germany, 2002.

[150] O. Kummer, F. Wienberg, and M. Duvigneau. Renew—The reference net workshop, online, URL: `http://renew.de/`, last visited: 10-11-2011.

[151] O. Kummer, F. Wienberg, and M. Duvigneau. Renew—User guide, online, URL: `http://www.informatik.uni-hamburg.de/TGI/renew/renew.pdf`, last visited: 10-11-2011.

[152] 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 J. Cortadella and W. Reisig, editors, *Applications and Theory of Petri Nets 2004*, volume 3099 of *Lecture Notes on Computer Science*, pages 484–493. Springer, 2004.

[153] L. A. Künzer. *Handlungsprädiktion zur Gestaltung einer adaptiven Benutzungsunterstützung in autonomen Produktionszellen.* Shaker, 2005.

[154] R. Lachman, J. L. Lachman, and E. C. Butterfield. *Cognitive Psychology and Information Processing.* Lawrence Erlbaum Associates, 1979.

[155] C. Lakos. From coloured Petri nets to object Petri nets. In G. de Michelis and M. Diaz, editors, *Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1995.

[156] P. Langley. *Elements of Machine Learning.* Morgan Kaufmann Publishers Inc., 1995.

[157] P. Langley. Machine learning for adaptive user interfaces. In *Proceedings of the 21st Annual German Conference on Artificial Intelligence: Advances in Artificial Intelligence*, KI'97, pages 53–62, London, UK, 1997.

[158] P. Langley. User modeling in adaptive interfaces. In *Proceedings of the 7th International Conference on User Modeling*, UMAP'99, pages 357–370, Secaucus, NJ, USA, 1999.

[159] M. E. Latoschik. A gesture processing framework for multimodal interaction in virtual reality. In *Proceedings of the 1st International Conference on Computer Graphics, Virtual Reality and Visualisation*, AFRIGRAPH '01, pages 95–100, Camps Bay, Cape Town, South Africa, 2001.

[160] M. E. Latoschik. Designing transition networks for multimodal VR-interactions using a markup language. In *Proceedings of the 4th IEEE International Conference on Multimodal Interfaces*, ICMI '02, pages 411–416, Washington, DC, USA, 2002.

[161] A. C. Lemke and G. Fischer. A cooperative problem solving system for user interface. In *Proceedings of the Association for the Advancement of Artificial Intelligence*, AAAI '90, pages 479–484, Boston, MA, USA, 1990.

[162] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[163] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1–2):181–224, 1993.

[164] M. Loy, R. Eckstein, D. Wood, J. Elliot, and B. Cole. *Java Swing.* O'Reilly Media, 2003.

[165] N. Luhmann. *Soziale Systeme. Grundriß einer allgemeinen Theorie.* Suhrkamp, 2001.

[166] M. T. Maybury and W. Wahlster. Intelligent user interfaces: An introduction. In *Proceedings of the 4th International Conference on Intelligent User Interfaces*, IUI '99, pages 3–4, Redondo Beach, CA, USA, 1999.

[167] G. McCluskey. Using Java Reflection, online, URL: `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`, last visited: 10-11-2011.

[168] R. McNaughton. *Elementary Computability, Formal Languages, and Automata.* Z B Pub Industries, 1993.

[169] Microsoft. Expression Blend 4, online, URL: `http://www.microsoft.com/expression/products/blend\_overview.aspx`, last visited: 10-11-2011.

[170] Microsoft. Microsoft Excel, online, URL: `http://office.microsoft.com/de-de/excel/`, last visited: 10-11-2011.

[171] C. A. Miller and R. Parasuraman. Designing for flexible interaction between humans and automation: Delegation interfaces for supervisory control. *Human Factors*, 49(1):57–75, 2007.

[172] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communictions*, 38(11):1917–1921, 1990.

[173] G. Mori, F. Paternò, and C. Santoro. Tool support for designing nomadic applications. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI '03, pages 141–148, Miami, FL, USA, 2003.

[174] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute of Technology, USA, 1981.

[175] M. Mühlenbrock and H. U. Hoppe. Computer supported interaction analysis of group problem solving. In *Proceedings of the Conference on Computer Supported Collaborative Learning*, CSCL '99, pages 398–405, Palo Alto, CA, USA, 1999.

[176] M. Mühlenbrock, F. Tewissen, and H. U. Hoppe. A framework system for intelligent support in open distributed learning environments. In *Proceedings of Artificial intelligence in Education: Knowledge and Media in Learning Systems*, AI-ED '97, pages 256–274, Kobe, Japan, 1997.

[177] B. A. Myers. A brief history of human-computer interaction technology. *ACM Interactions*, 5(2):44–54, 1998.

[178] M. Nagl. A tutorial and bibliographical survey on graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and their Application to Computer Science and Biology*, volume 73 of *Lecture Notes of Computer Science*, pages 70–126. Springer, 1978.

[179] D. Navarre, P. Palanque, and S. Basnyat. A formal approach for user interaction reconfiguration of safety critical interactive systems. In M. Harrison and M.-A. Sujan, editors, *Computer Safety, Reliability, and Security*, volume 5219 of *Lecture Notes in Computer Science*, pages 373–386. Springer, 2008.

[180] D. Navarre, P. Palanque, R. Bastide, A. Schyn, M. Winckler, L. Nedel, and C. Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In M. Costabile and F. Paternò, editors, *Human-Computer Interaction—INTERACT 2005*, volume 3585 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2005.

[181] D. Navarre, P. Palanque, R. Bastide, and O. Sy. Structuring interactive systems specifications for executability and prototypability. In P. Palanque and F. Paternò, editors, *Interactive Systems Design, Specification, and Verification*, volume 1946 of *Lecture Notes in Computer Science*, pages 97–119. Springer, 2001.

[182] 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*, 16(4):1–56, 2009.

[183] 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 T. Graham and P. Palanque, editors, *Interactive Systems. Design, Specification, and Verification*, volume 5136 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2008.

[184] A. Newell and S. K. Card. The prospects for psychological science in human-computer interaction. *Human-Computer Interaction*, 1(3):209–242, 1985.

[185] M. Newman. *Networks: An Introduction*. Oxford University Press, 2010.

[186] W. M. Newman. A system for interactive graphical programming. In *Proceedings of the Joint Computer Conference*, AFIPS '68, pages 47–54, Atlantic City, NJ, USA, 1968.

[187] R. R. Nickerson. On the distribution of cognition: Some reflections. In G. Salomon, editor, *Distributed cognitions: Psychological and educational considerations*, chapter 8. Cambridge University Press, 1997.

[188] D. A. Norman. Design principles for human-computer interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, pages 1–10, Boston, MA, USA, 1983.

[189] A. S. Nowak and R. I. Carr. Classification of human errors. In *Proceedings of the Symposium on Strucutral Safety Studies*, pages 1–10, Denver, CO, USA, 1985.

[190] OASIS. User Interface Markup Language (UIML) Specification, online, URL: `http://www.oasis-open.org/committees/uiml/`, last visited: 10-11-2011.

[191] OASIS. Web Services Business Process Execution Language, Version 2.0: OASIS Standard, online, URL: `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, last visited: 10-11-2011.

[192] H. Oberheid and D. Söffker. Cooperative arrival management in air traffic control—A coloured petri net model of sequence planning. In *Proceedings of the 8th International Conference on Application of Concurrency to System Design*, ACSD'08, pages 348–367, Xian, China, 2008.

[193] J. O'Brien and M. Shapiro. An application framework for collaborative, nomadic applications. Technical report, INRIA, 2006.

[194] D. R. Olsen. Propositional production systems for dialog description. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Empowering People*, CHI '90, pages 57–64, Seattle, WA, USA, 1990.

[195] OMG. The current official UML specification, online, URL: `http://www.omg.org/technology/documents/modeling\_spec\_catalog.htm\#UML`, last visited: 10-14-2011.

[196] OMG. UML specification 2.4, online, URL: `http://www.omg.org/spec/UML/2.4/`, last visited: 10-11-2011.

[197] OMG. Updated BPMN 2.0 specification, online, URL: `http://www.omg.org/cgi-bin/doc?dtc/10-06-04`, last visited: 10-11-2011.

[198] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen.* Spektrum Akademischer Verlag, 5th edition, 2012.

[199] J. Ou, S. R. Fussell, X. Chen, L. D. Setlock, and J. Yang. Gestural communication over video stream: Supporting multimodal interaction for remote collaborative physical tasks. In *Proceedings of the 5th International Conference on Multimodal Interfaces*, ICMI '03, pages 242–249, Vancouver, British Columbia, Canada, 2003.

[200] S. Oviatt. Ten myths of multimodal interaction. *Communications of ACM*, 42(11):74–81, 1999.

[201] P. Palanque. *User-Driven user interfaces modeling using interactive cooperative objects.* PhD thesis, University Toulouse I, France, 1992.

[202] P. Palanque and R. Bastide. Petri net based design of user-driven interfaces using the interactive cooperative objects formalism. In F. Paternò, editor, *Interactive Systems: Design, Specification, and Verification*, pages 383–400. Springer, 1994.

[203] P. Palanque and R. Bastide. Formal specification and verification of CSCW using the interactive cooperative object formalism. In *Proceedings of 10th Conference on People and Computers*, HCI '95, pages 213–231, Tokyo, Japan, 1995.

[204] P. Palanque and F. Paternò. *Formal Methods in Human-Computer Interaction.* Springer, 1998.

[205] R. Parasuraman and V. Riley. Humans and automation: Use, misuse, disuse, abuse. *Human Factors*, 39(2):230–253, 1997.

[206] 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*, 30(3):286–297, 2000.

[207] A. J. Parkin. *Essential Cognitive Psychology.* Psychology Press, 2000.

[208] D. L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 24th ACM National Conference*, ACM '69, pages 379–385, New York, NY, USA, 1969.

[209] F. Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer London, UK, 1999.

[210] F. Paternò. Model-based design of interactive applications. *Magazine Intelligence*, 11(4):26–38, 2000.

[211] F. Paternò and C. Santoro. One model, many interfaces. In C. Kolski and J. Vanderdonckt, editors, *Computer-Aided Design of User Interfaces III*, chapter 13. Kluwer Academic Publishers, 2002.

[212] J. F. Patterson, R. D. Hill, S. L. Rohall, and S. W. Meeks. Rendezvous: An architecture for synchronous multi-user applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, CSCW '90, pages 317–328, Los Angeles, CA, USA, 1990.

[213] W. Paulus. SwiXML, online, URL: `http://www.swixml.org/index.html`, last visited: 10-11-2011.

[214] S. J. Payne and T. R. Green. Task-action grammars: A model of the mental representation of task languages. *Human-Computer Interaction*, 2(2):93–133, 1986.

[215] P. Peltonen, E. Kurvinen, A. Salovaara, G. Jacucci, T. Ilmonen, J. Evans, A. Oulasvirta, and P. Saarikko. It's mine, don't touch!: Interactions at a large multi-touch display in a city centre. In *Proceeding of the 26th Annual SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1285–1294, Florence, Italy, 2008.

[216] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Germany, 1962.

[217] G. E. Pfaff, editor. *User Interface Management Systems*. Springer, 1983.

[218] C. Phanouriou. *UIML: A Devie-Independent User Interface Markup Language*. PhD thesis, Virgina Polytechnic Institute and State University, USA, 2000.

[219] J. Preece. *A Guide to Usability*. Addison-Wesley, 1993.

[220] L. Priese and H. Wimmel. *Petri-Netze*. Springer, 2008.

[221] J. Protzenko. *XUL. Entwicklung von Rich Clients mit der Mozilla XML User Interface Language*. Open Source Press, 2007.

[222] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.

[223] A. R. Puerta, E. Cheng, T. Ou, and J. Min. MOBILE: User-centered interface building. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: The CHI is the Limit*, CHI '99, pages 426–433, Pittsburgh, PA, USA, 1999.

[224] J. Raskin. *The Humane Interface*. Addison Wesley, 11th edition, 2000.

[225] J. Rasmussen. Human errors. A taxonomy for describing human malfunction in industrial installations. *Journal of Occupational Accidents*, 4(2–4):311–333, 1982.

[226] J. Rasmussen. Skills, rules, knowledge: Signals, signs, and symbols, and other distinctions in human performance models. *IEEE Transactions on Systems, Man, Cybernetics*, 13(3):257–267, 1983.

[227] J. Rasmussen. Mental models and the control of action in complex environments. In D. Ackermann, D. Tauber, and M. J. Tauber, editors, *Mental Models and Human-Computer Interaction*, pages 41–70. Elsevier, 1990.

[228] M. Rauterberg, S. Schluep, and M. Fjeld. How to model behavioural and cognitive complexity in human-computer interaction with Petri nets. In *Proceedings of 6th IEEE International Workshop on Robot and Human Communication*, RO-MAN '97, Sendai, Japan, 1997.

[229] L. Razmerita, A. Angehrn, and A. Maedche. Ontology-based user modeling for knowledge management systems. In *Proceedings of the International Conference on User Modeling*, volume 2702 of *Lecture Notes in Computer Science*, pages 213–217. Springer, 2003.

[230] L. M. Reeves, J. Lai, J. A. Larson, S. Oviatt, T. S. Balaji, S. Buisine, P. Collings, P. Cohen, B. Kraal, J.-C. Martin, M. McTear, T. V. Raman, K. M. Stanney, H. Su, and Q. Y. Wang. Guidelines for multimodal user interface design. *Communications of ACM*, 47(1):57–59, 2004.

[231] J. Reimer. A history of the GUI, online, URL: `http://arstechnica.com/old/content/2005/05/gui.ars/`, last visited: 10-11-2011.

[232] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[233] U. Rembold and P. Levi. *Einführung in die Informatik für Naturwissenschaftler und Ingenieure*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 2002.

[234] G. D. Rey and K. F. Wender. *Neuronale Netze*. Huber, 2011.

[235] M. Ringel, A. M. Piper, A. Cassanego, and T. Winogard. Supporting Cooperative Language Learning: Issues in Interface Design for an Interactive Table, online, URL: `http://hci.stanford.edu/cstr/reports/2005-08.pdf`, last visited: 10-11-2011.

[236] S. P. Robertson and J. B. Black. Planning units in text editing behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, pages 217–221, Boston, MA, USA, 1983.

[237] G. Rosenzweig. *ActionScript 3.0 Game Programming University*. Que, 2nd edition, 2011.

[238] M. B. Rosson and J. M. Carroll. *Usability Engineering*. Kaufmann, 2006.

[239] W. B. Rouse and N. M. Morris. On looking into the black box: Prospects and limits in the search for mental models. *Psychological Bulletin*, 100(3):349–363, 1986.

[240] N. B. Sarter and D. D. Woods. How in the world did we ever get into that mode? Mode error and awareness in supervisory control. *Human Factors*, 37(1), 1995.

[241] N. B. Sarter, D. D. Woods, and C. E. Billings. Automation surprises. In G. Salvendy, editor, *Handbook of Human Factors & Ergonomics*, pages 1–25. Wiley, 2nd edition, 1997.

[242] O. Schmitz. Understanding algorithms and abstract data structures by means of interface configuration. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2011.

[243] M. Schneider-Hufschmidt, T. Kühme, and U. Malinowsk. *Adaptive user interfaces: Principles and practice*, volume 10 of *Human Factors in Information Technology*. North-Holland, 1993.

[244] U. Schöning. *Logik für Informatiker*. Spektrum, 5th edition, 2000.

[245] C. Schubert. Dynamische Schnittstellenrekonfiguration im Rahmen einer verteilten Simulationsumgebung. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2007.

[246] A. Schür and B. Westfechtel. Graph grammars and graph rewriting systems. Technical report, RWTH Aachen, 1992.

[247] A. Sears and J. A. Jacko. *The Human-Computer Interaction Handbook*. Erlbaum, 2nd edition, 2008.

[248] L. Selvanadurajan. Interaktive Visualisierung kryptographischer Protokolle mit Concept Keyboards—Testszenarien und Evaluation. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2007.

[249] C. A. Shaffer, M. Cooper, and S. H. Edwards. Algorithm visualization: A report on the state of the field. *ACM SIGCSE Bulletin*, 39(1):150–154, 2007.

[250] E. Shakshuki, A. Kerren, and T. Muldner. Web-based structured hypermedia algorithm explanation system. *International Journal of Web Information Systems*, 3(3):179–197, 2007.

[251] S. A. Shappell and D. A. Wiegmann. Applying reason—the human factors analysis and classification system (HFACS). *Human Factors and Aerospace Safety*, 1(1):59–86, 2001.

[252] H. Sharp, Y. Rogers, and J. Preece. *Interaction Design*. Wiley, 2nd edition, 2007.

[253] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology*, 1(3):237–256, 1982.

[254] B. Shneiderman and C. Plaisant. *Designing the User Interface*. Addison-Wesley, 5th edition, 2010.

[255] A. Simon and S. Scholz. Multi-viewpoint images for multi-user interaction. In *Proceedings of IEEE Virtual Reality*, VR '05, pages 107–113, Bonn, Germany, 2005.

[256] D. Söffker. *Systemtheoretische Modellbildung der wissensgeleiteten Mensch-Maschine-Interaktion*. Logos, 2003.

[257] I. Sommerville. *Software Engineering*. Pearson Studium, 8th edition, 2007.

[258] D. Sonntag, R. Engel, G. Herzog, A. Pfalzgraf, N. Pfleger, M. Romanelli, and N. Reithinger. Smartweb handheld—multimodal interaction with ontological knowledge bases and semantic web services. In T. Huang, A. Nijholt, M. Pantic, and A. Pentland, editors, *Artifical Intelligence for Human Computing*, volume 4451 of *Lecture Notes in Computer Science*, pages 272–295. Springer, 2007.

[259] G. Stahl, T. Koschmann, and D. Suthers. Computer-supported collaborative learning: An historical perspective. In R. K. Sawyer, editor, *Cambridge handbook of the learning sciences*, pages 409–426. Cambridge University Press, 2008.

[260] A. Stanciulescu. *A methodology for developing multimodal user interfaces of information systems*. PhD thesis, University of Louvain, France, 2008.

[261] J. T. Stasko. TANGO: A framework and system for algorithm animation. *SIGCHI Bulletin*, 21(3):59–60, 1990.

[262] J. T. Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, 1992.

[263] J. T. Stasko. *Software visualization: Programming as a multimedia experience*. MIT Press, 1998.

[264] R. D. Stevens, A. D. Edwards, and P. A. Harling. Access to mathematics for visually disabled students through multimodal interaction. *Human-Computer Interaction*, 12(1):47–92, 1997.

[265] J. Stückrath. Inkrementelle Interaktionsmodellierung mit farbigen Petri-Netzen—Formale Beschreibung und Implementierung. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2010.

[266] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principals and Paradigms*. Prentice Hall International, 2nd edition, 2006.

[267] J. Tidwell. *Designing Interfaces*. O'Reilly, 2005.

[268] M. Trompedeller. A classification of Petri nets, online, URL: `http://www.informatik.uni-hamburg.de/TGI/PetriNets/classification/`, last visited: 10-11-2011.

[269] E. Tulving. Episodic and semantic memory. In E. Tulving and W. Donaldson, editors, *Organisation of Memory*, pages 381–403. Academic Press, 1972.

[270] E. Tulving. *Elements of Episodic Memory*. Oxford University Press, 1983.

[271] J. Urquiza-Fuentes and J. Velázquez-Iturbide. A survey of successful evaluations of program visualization and algorithm animation systems. *ACM Transactions on Computing Education*, 9(2):1–21, 2009.

[272] E. van der Vlist. *Relax NG*. O'Reilly, 2004.

[273] J. Vanderdonckt, Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan, and M. Florins. UsiXML: A user interface description language for specifying multimodal user interfaces. In *Proceedings of W3C Workshop on Multimodal Interaction*, WMI '04, pages 35–42, Sophia Antipolis, France, 2004.

[274] W3C. HTML5: A vocabulary and associated APIs for HTML and XHTML, online, URL: `http://www.w3.org/TR/html5/`, last visited: 10-11-2011.

[275] W3C. XSL Transformations (XSLT) Version 2.0, online, URL: `http://www.w3.org/TR/xslt20/`, last visited: 10-11-2011.

[276] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *Proceedings of International Conference on Automated Software Engineering*, ASE '06, pages 37–46, Los Alamitos, CA, USA, 2006.

[277] N. M. Webb. Peer interaction and learning in cooperative small groups. *Journal of Educational Psychology*, 74(5):642–655, 1982.

[278] N. M. Webb. Student interaction and learning in small groups. *International Journal of Educational Research*, 52(3):421–445, 1982.

[279] M. Weber and E. Kindler. The Petri net markup language. In *Petri Net Technology for Communication-Based Systems—Advances in Petri Nets*, volume 2472 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 2003.

[280] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A graph based architectural (re)configuration language. *SIGSOFT Software Engineering Notes*, 26(5):21–32, 2001.

[281] B. Werther. *Kognitive Modellierung mit Farbigen Petrinetzen zur Analyse menschlichen Verhaltens*. PhD thesis, Technical University of Braunschweig, Germany, 2006.

[282] B. Weyers. An error-driven approach for automated user-interface redesign—Concepts and architecture. In N. Baloian, W. Luther, D. Söffker, and Y. Urano, editors, *Proceedings of the DAAD Summer University Chile*, pages 104–113. Logos, 2008.

[283] B. Weyers, N. Baloian, and W. Luther. Cooperative creation of concept keyboards in distributed learning environments. In *Proceedings of 13 th International Conference on CSCW in Design*, CSCWD '09, pages 534–539, Santiago, Chile, 2009.

[284] B. Weyers, D. Burkolter, A. Kluge, and W. Luther. User-centered interface reconfiguration for error reduction in human-computer interaction. In *Proceedings of the 3rd International Conference on Advances in Human-Oriented and Personalized Mechanisms, Technologies, and Services*, CENTRIC '10, pages 50–55, Nizza, France, 2010.

[285] 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*, accepted, 2012.

[286] B. Weyers and W. Luther. Formal modeling and reconfiguration of user interfaces. In *Proceedings of Jornadas Chilenas de Computacion*, JCC '10. IEEE, Antofagasta, Chile, 2010.

[287] B. Weyers and W. Luther. Formal modelling and identification of operating errors for formal user interface reconfiguration. In *Proceedings of 7th Vienna Conference on Mathematical Modelling*, MATHMOD '12, Vienna, Austria, 2012.

[288] B. Weyers, W. Luther, and N. Baloian. Cooperative model reconstruction for cryptographic protocols using visual languages. In L. Carrico, N. Baloian, and B. Fonseca, editors, *Groupware: Design, Implementation, and Use CRIWG 2009*, volume 5784 of *Lecture Notes in Computer Science*, pages 311–318. Springer, 2009.

[289] B. Weyers, W. Luther, and N. Baloian. Interface creation and redesign techniques in collaborative learning scenarios. *Future Generation Computer Systems*, 27(1):127–138, 2011.

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

[291] C. Wickens. Cognitive factors in aviation. In R. S. Nickerson, editor, *Handbook of Applied Cognition*, pages 247–282. Wiley, 1999.

[292] C. Wickens and J. G. Hollands. *Engineering Psychology and Human Performance*. Prentice Hall, 3rd edition, 2000.

[293] A. Wiegmann and A. Shappell. *A Human Error Approach to Aviation Accident Analysis*. Ashgate, 2003.

[294] D. Wigdor, G. Penn, K.y Ryall, A. Esenther, and C. Shen. Living with a tabletop: Analysis and observations of long term office use of a multi-touch table. In *Proceedings of International IEEE Workshop on Horizontal Interactive Human-Computer Systems*, TABLETOP '07, pages 60–67, Newport, RI, USA, 2007.

[295] J. C. Wileden. Relationship between graph grammars and the design and analysis of concurrent software. In G. Goos and J. Hartmanis, editors, *Proocedings of the International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 456–463. Springer, 1978.

[296] P. Wilke. Zusammenhänge und Unterschiede zwischen Graph-Grammatiken und Petri Netzen sowie verwandter Systeme. Master's thesis, University of Erlangen, Department of Computer Science, Germany, 1983.

[297] J. Williamson, R. Murray-Smith, and S. Hughes. Shoogle: Excitatory multimodal interaction on mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 121–124, San Jose, CA, USA, 2007.

[298] J. R. Wilsong and A. Rutherford. Mental models: Theory and application in human factors. *Human Factors*, 31(6):617–634, 1989.

[299] D. D. Woods and N. B. Sarter. Learning from automation surprises and "going sour" accidents. In N. B. Sarter and R. Amalberti, editors, *Cognitive Engineering in the Aviation Domain*, pages 327–368. CRC Press, 2000.

[300] UIMS Tool Workshop. A metamodel for the runtime architecture of an interactive system. *SIGCHI Bulletin*, 24(1):32–37, 1992.

[301] Petri Net World. Petri net world community website, online, URL: `http://www.informatik.uni-hamburg.de/TGI/PetriNets/`, last visited: 10-11-2011.

[302] R. M. Young. The machine inside the machine: Users' models of pocket calculators. *International Journal of Man-Machine Studies*, 15(1):51–85, 1981.

[303] L. Yu. Regelbasierte automatische Rekonfiguration der Benutzerschnittstelle aus formal beschriebenen Bedienungsfehlern. Master's thesis, University of Duisburg-Essen, Department of Computer Science and Applied Cognitive Science, Germany, 2011.

[304] G. Zurita, N. Baloian, and F. Baytelman. A collaborative face-to-face design support system based on sketching and gesturing. *Advanced Engineering Informatics*, 22(3):340–349, 2008.