A Component-Based Approach to Human-Computer Interaction

Specification, Composition, and Application to Information Services

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Thomas Feyer
geboren am 14. September 1972 in Halle

Gutachter: Prof. Dr. rer. nat. habil. Bernhard Thalheim
Gutachter: Prof. Mag. Dr. rer. nat. Dr. h. c. Heinrich C. Mayr
Gutachter: Prof. Dr.-Ing. Hannu Jaakkola

Tag der mündlichen Prüfung: 11. Dezember 2003

To my father.

Abstract

The discipline of software engineering is increasingly shifting from classical design and development tasks towards tasks concerning reuse, adaptation, and integration. Driving motivations for this shift are (i) decreasing development time and costs and (ii) increasing quality of design results. Unfortunately, these new tasks are not yet supported sufficiently. While classical approaches to information system's design are quite well suited to a design from scratch, they do not provide powerful concepts concerning reuse. Although encapsulation by means of functions, classes, or sub-systems provides opportunities for reusing functionality, methods concerning encapsulation of dialog structures are commonly neglected. On the one hand, an approach to reusing dialog structures seems quite promising wrt. design time and quality, since interaction design has been recognized as a complex and time-consuming task. On the other hand, this approach reveals an inherent complexity which must be resolved. It manifests at the following questions:

- How to encapsulate interaction by means of components?
- Does component composition enable to derive complex interaction from elementary?
- How to verify (dynamic) properties of components wrt. quality of user interaction?
- How to ensure that a composition of components behaves as intended?
- How to adapt components to particular application requirements?
- Which new requirements does a component-based design process have to meet? In particular:
- How to support the designer in identifying desired components?
- How to identify relations between components as, for example, component refinement?

To tackle solutions to these questions, we identified three closely related research areas as a basis: (i) interaction patterns, (ii) interaction specification and design, and (iii)

component approaches. At the thesis, we selected promising models of these areas, adapted them to our requirements, and provided an integrated treatment which combines the advantages of all of these approaches. Therewith, we hope that the results of the thesis will contribute to the currently active research concerning component technology and methods of reuse.

Acknowledgments

This work has been developed within the last years at the database and information system's (dbis) group at BTU Cottbus. I would like to thank everybody who was contributing to this work by discussions, support, as well as diversions. Especially, I would like to express my gratitude to the following people:

First of all, I like to thank my advisor Prof. Bernhard Thalheim for his support, inspirations, and hospitality. For their effort to provide the reviewing, I like to thank Prof. Heinrich Mayr and Prof. Hannu Jaakkola. Particularly, I thank the "pre-reviewers" Steffen Jurk, Aleksander Binemann-Zdanovicz, Gunar Fiedler, and Vojtech Vestenický who contributed to the final appearance of the thesis. I would like to thank Wolfram Clauß, Jana Lewerenz, Srinath Srinivasa, and the members of the former codesign team for many fruitful discussions at the initial stage of the thesis. Regarding the initial stage, I will not miss to thank Prof. Klaus-Dieter Schewe for paving the way to the BTU. A special thank I would like to send to Marcela Varas and her group at the Universidad de Concepción. Thank you for the interesting discussions and the perfectly arranged stay at your group.

I also thank all students which I was allowed to advise — especially to Birk Heinze for many discussions which provided essential ideas for this work. For discussions as well as welcome diversions, I like to thank all former and current members of the dbis team, in particular, our secretary Karla Kersten for her help at any incidents. Troubles related to system administration were always fixed by Günter Millahn and Thomas Kobienia. Thanks a lot.

Last but not least, I greatly thank my parents and my wife Christiane for all their support and motivation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Design of adaptive and ergonomic user interfaces has been recognized as a complex and expensive task [Nie94, DFAB98, HLP97, NL95]. In particular, at information services available to a large user variety, the acceptance of a system strongly depends on the adequateness of its user interface. Primarily at the advent of the Web, a large number of information services were developed and made available to public access. At an early phase (but partly valid today), usability of user interfaces was often circumstantial, inconsistent, and barely intuitive. However, through a successive evolution, specific dialog structures were identified which proved to enable efficient and intuitive interaction. They were employed to support, for example, navigation and search scenarios, personalization, help facilities, or human error handling. This evolution story was not only observable according to new challenges of the World-Wide Web. For example, menu-like interaction structures have been discovered by a similar process decades ago. When dialog structures were recognized to support ergonomic interaction, they were widely reused. However, reuse was commonly realized in an ad-hoc manner, i.e., its realization was basically guided by the visual part of user interaction only. To improve the process of reuse, an abstraction of so-called *interaction pattern* was introduced [Bor01, Tid98, WT00a, WT00b, DLH02, GPBV99, RSL00] — motivated by the general notion of design patterns [Ale79, GHJV93].

Instead of concrete visualizations, interaction patterns propose guidelines to solve certain interaction problems. Rather simple patterns that support navigation and search are known as (cf. [RSL00, HDP02, Tid98, GC00]):

Set-Based Navigation allowing a user to browse through a collection of elements (typically resulting from a previous search),

Interaction History allowing a user to survey activities performed beforehand and to return to previous dialog situations,

Guided Tour providing a user with a brief overview of the content and/or services provided by the system, and

Selectable Search Space allowing a user to decrease large search spaces by choosing specific categories (comparable to slice and dice facilities of OLAP interfaces [AGS97, GL97, LST99]).

Although these examples exclusively represent domain independent patterns, an application of the approach to domain specific patterns is beneficial as well. For example, a pattern describing required interactivity of a shopping cart is rather dedicated to commercial domains.

Quality of Service

Today, services provided by human service providers as, for example, travel agencies, book stores, or dressing shops, often still exceed their according automated services based on human-computer interaction in (i) effectiveness, (ii) quality of results, and (iii) user (customer) satisfaction. Reasons therefore are manifold. Human dialogs can be understood as interactive scenarios, i.e., sequences of utterances of different actors. For each actor, a dialog usually requires performing several (sub-)tasks to achieve an intended goal. According to accomplishing a task, actors apply different behavior patterns which they learned to be effective to achieve the particular task. Examples of abstract patterns at the area of service provision are: (i) greeting, (ii) expressing a wish, (iii) expressing/changing requirements, (iv) asking for clarification, (v) performing transactions, or (vi) aborting transactions.

One reason for lower effectiveness of currently provided user interfaces of (automated) information services lies in their restricted opportunities of interaction. At human interaction, behavior patterns may be combined flexibly at different dialog situations. In contrast, modeling of human-computer interaction commonly considers specific interactive paths only which strongly restricts interactive abilities. Interaction patterns provide a promising candidate to improve this situation. Firstly, adequate realizations of behavior patterns by means of human-computer interaction may be provided by interaction patterns. Since behavior patterns might correspond to several realizations, it is the task of the designer to choose appropriate interaction patterns. Secondly, a notion of pattern composition must be supported. It therewith permits a user to flexibly combine their interactive facilities.

1.1 Interaction Patterns

According to the formulation of interaction patterns, informal descriptions are used which commonly comprise a pattern name, a problem statement, forces, examples, solutions, and relations to other patterns (cf., for example, [Bor01, Tid98]):

Name uniquely identifies an interaction pattern within a pattern repository. There exists several guidelines, how pattern names should be selected. Generally, the

name should provide a short and comprehensible abstraction of the pattern. In addition to a repository structuring, it supports retrieval of patterns.

Problem statement describes the general interactive problem which will be resolved. It may also indicate the situations in which the interaction pattern can be applied (sometimes also referred to as 'Context').

Forces discuss existing constraints which must be considered at possible solutions. For example, a constraint may consider the boundedness of short-term memory of human users. Possible solutions then must take this constraint into account, for example, by repeating information at subsequent dialog steps. In addition, forces may provide indications why the pattern is meaningful in certain situations.

Examples provide exemplary solutions of the pattern at concrete situations. They are useful to provide an understanding of the construction of solution according to the problem.

Solutions describe opportunities how the interactive problem can be resolved. Thereby, it provides a generalization of given examples and may suggest design recommendations. Besides a description, schematic diagrams may be used to illustrate solutions. Formality of diagrams may reach from simple sketches to formal specifications.

Related patterns (sometimes also referred to as 'References' or 'Context') indicate associations to other patterns within a pattern repository. Common associations are specialization and aggregation. According to aggregation, suggestions are proposed which patterns may be used for composition.

As an example, consider the following formulation of interaction pattern [Interaction History]. It represents an abbreviated version of the interaction pattern introduced in [Tid98]. (A detailed version of this pattern and others can be found in Appendix A.)

Interaction pattern: [Interaction History]

Problem statement

A user performs a sequence of actions with an artifact, or navigates through it. Should the artifact keep track of what the user does with it? If so, how?

Forces

- People are forgetful of tedious details; users are not likely to remember just what they've recently done with the artifact, and computers are better at it than people are.

- User may need to know exactly what they have just done, so they can undo their work or backtrack.
- Users may want a high-level overview of what they have done, to gain understanding that they wouldn't get just from memory.
- Audit logs are sometimes necessary, such as with legal regulatory requirements.
- Highly interactive artifacts may generate huge amounts of recordable detail.

Examples

- The "history" or "visited links" feature on a Web browser
- UNIX shell's saved command history
- Logs of exchanged email or other social exchanges

Solution

Record the sequence of interactions as a "history". Keep track of enough detail to make the actions repeatable, scriptable, or even undoable, if possible. Provide a comprehensible way to display the history to the user; most artifacts that implement this pattern use a textual representation, especially [Composed Command], but that's not a requirement. (In fact, a history for [Navigable Spaces] may be better portrayed as a state diagram, showing single steps, backtracks, etc.) If the artifact is capable of saving its state, as with [Remembered State], give the user the option of saving the history from session to session.

Having a history around provides users with a set of milestones that they can use with Go Back to a Safe Place – but explicitly think about whether you want to actually undo all the history between the "present" and the point in the history that the user wants to fall back to. The answer will depend upon your specific circumstances.

Related Patterns

The pattern may be used together with [Navigable Spaces], [Control Panel], [WYSIWYG Editor], [Composed Command], and [Social Space].

1.2 Towards Composable Specifications

While the pattern approach relieves the task of finding and realizing appropriate solutions, it still suffers several shortcomings:

(i) Some of the approaches as, for example, [DLH02, GPBV99, RSL00], primarily target Web interfaces. As interface paradigms change over time, interface independence is an essential factor for long-term use of patterns (cf. also [SF02]).

(ii) Since interaction patterns are described informally, it is difficult to verify properties according to their quality.

(iii) For the same reason, it is difficult to verify, whether a concrete realization (or instantiation) of a pattern complies with its intention.

(iv) Composition of patterns is considered in a rudimentary way only, for example, by verbal description of relations to other patterns. However, elementary interaction patterns commonly cannot comprise the specification of a complete system, but only approved sub-structures. To reuse interactive structures for larger systems as well, compositionality plays an indispensable role. For example, interactive catalogs support the user by flexible navigation structures to search and explore large, categorized collections. Thus, a catalog pattern comprises several elementary patterns as, for example, [Set-Based Navigation], [Interaction History], and [Selectable Search Space]. At current approaches, an opportunity to verify, if (and how) instantiations of patterns can be composed is missing.

This thesis was inspired by the idea of reusing ergonomic interactive structures in the context of data intensive information services. To enable verification of the quality of interaction as well as a meaningful composition, we have decided in favor of a formal model. There exist several formal models to specify interactive system behavior. Examples of interface independent models are state transition networks (STN) [New68, Par69, Was85], state charts [HP98], flow charts, petri nets [BP95, CL99], grammars as BNF or production systems, communicating sequential processes (CSP) [Ale87, Abo90], or calculi [Sri01]. There also exist interface dependent approaches as, for example, models dedicated to graphical user interfaces (GUI) [BFJ96], or models dedicated to Web interfaces [SR98, MAM03, FST98, FP98, FKST00, GC99, FT99, GGS$^+$99b, HF99].

We employed Coloured Petri nets (CP nets) as a basis for the component model proposed in the thesis because of the following reasons:

- The granularity of input and output elements can be varied by the designer. This allows to model interaction at different levels of abstraction.
- According to composition, CP nets offer a natural framework to model parallelism and synchronization. Net specifications circumvent the problem of combinatorial explosion in case of concurrent dialogs. This problem is known from state transition approaches, if concurrent events as, for example, 'escape' facilities have to be added. Thus, concurrency is an essential feature according to compositionality. Otherwise, composition of dialog structures is rather limited to the sequential case.

- Specifications may be independent from particular user interface paradigms. In fact, depending on the granularity of user events and system responses, specifications can be completely independent from, dependent on a class of, or dependent on specific user interface paradigms (as, for example, HTML). Throughout the thesis, we mainly concern interface independent specifications, since they address a larger class of applications, in particular, future user interface paradigms.

- Besides the opportunity to model user interaction, CP nets may specify interactive behavior in general, and thus, interaction with databases or networks in particular. According to the domain of data intensive information services, database interaction plays an important role.

- There exist several opportunities for functional abstraction and specialization, for example, by refining transitions, arc expressions, or data types.

- CP nets possess a well-defined semantics. It permits to prove dynamic properties and can be used to validate qualitative aspects of user interaction.

- There exist graphical representations.

To encapsulate interactive behavior, we introduce the following abstractions. Firstly, we propose *interaction nets* [FT02b] which extend CP nets by interfaces and the facility of composition. Interaction nets can be interpreted as white-box components as their ”implementation” is known. The behavior of interaction nets can be analyzed by standard methods of place/transition nets (cf. [Jen97a, GV02]). Secondly, we derive a black-box semantics of interaction nets [FT03] which corresponds to the component framework proposed by Broy et. al. [BS01]. There, components are considered as black-boxes with (i) an external interface and (ii) a behavior specification which determines the relation between input streams and output streams. Thereby, a stream represents a finite or infinite sequence of messages. We show that interaction net composition and component composition coincide with one another wrt. their external behavior. It opens the opportunity to combine the advantages of net-based specifications with those of component approaches. These include:

(i) Interaction specification and verification based on a net formalism.

(ii) Verification of dynamic properties of composite components, even in the case that (i) some sub-components are not net specifications, or (ii) the precise net specification of sub-components is not known.

(iii) Specification of dynamic interface assertions. Besides classical type constraints, interfaces may be augmented by assertions which must be obeyed by other components within a composition. It yields a so-called ”intended” behavior, since properties of composite behavior can be verified locally (i.e., in advance). In

other words, components that violate required assertions are excluded, since they yield composite behavior which cannot be anticipated and might be undesired. In particular, this opportunity circumvents considerable costs of (re-)analyzing composite net specifications, after their sub-nets have been analyzed already. Instead, properties of the composition can be derived from properties of its sub-components.

(iv) A specialization relation proposed for the component framework can be carried over to net-based specifications. Thereby, a hierarchical collection can be provided for later reuse.

To practically apply the formal interaction model to the specification of user interaction at the area of information services, we introduce an abstraction called *ui-components*. They are founded on interaction nets and permit to specify dialog structures at a more user-oriented level. Particularly, they are used to derive complex dialog structures from elementary ones by composition. Thereby, dependencies between dialog structure are specified explicitly. As composition of ui-components scales up to composite ui-components of any complexity, they are intended to provide a repository of dialog structures satisfying particular purposes for later reuse. Commonly, dialog structures at an elementary level should be rather domain-independent — corresponding to interaction patterns as, for example, [Set-Based Navigation] or [Interaction History]. Based on them, higher-level components provide domain adaptation by deriving composite and adapted dialog structures from elementary ones — corresponding to interaction patterns as, for example, [Shopping Cart]. Note that besides dialog structures, ”adaptations” can explicitly be represented by separate components. Thereby, they permit adaptations of existing dialogs to requirements of particular tasks and users.

The described concepts are introduced at the thesis by separate chapters. The concept of interaction nets and their composition is defined in Chapter 2. It also motivates, how they can be utilized for interaction design. Chapter 3, develops an opportunity to consider interaction nets as component specifications. More precisely, we propose an embedding of interaction nets into an existing component model. As an appropriate component model for this task, we identified the model proposed by Broy et. al. [BS01]. It is briefly introduced at the beginning of Chapter 3. At the end of the chapter, we discuss semantic issues of composition. While at a syntactical level, composition is clearly defined by means of interface matching based on specified data types, at a semantical level, composition is unrestricted. In other words, although components can be composed syntactically, the behavior of the composition might be unintended. As explained above, we introduce the notion of *(interface) assertions* which restrict composition at a semantical level as well. It resembles the notion of *contracts* known from component approaches. Afterwards, Chapter 4 introduces the practical application of above theory. It proposes the abstraction of ui-components and provides example cases, how dialog structures and their (dependent) compositions can be specified. The chapter also includes an approach to verify properties of interface quality. Thereby,

we focus on selected questions according to ergonomics of dialog structures. For example, we consider questions like ”Do dialog specifications (based on ui-components) correspond to a given task model wrt. completeness and correctness?”

Relations of the introduced approach to other work is presented in Chapter 5. Thereby, we particularly consider formal design approaches to information services and component approaches. While we less emphasize requirements and pragmatism of a component-based design approach throughout the thesis, we present a discussion about related issues there. Finally, we conclude the thesis in Chapter 6 by summarizing the achievements and indicating open work.

Chapter 2

A Net-Based Interaction Model

An elementary issue of (re-)using dialog structures consists of their integration. As sequential composition significantly simplifies this issue, components are often considered as closed sub-systems that comprise an autonomous specification (or implementation) of required interactive behavior. On the one hand, this approach does not require a sophisticated model for composition. On the other, its flexibility is limited, since extensions or adaptations, in particular, according to user interaction can hardly be incorporated. In this chapter, we introduce a formal model which permits interleaved composition. Thereby, adaptation of interactive behavior as well as resolution of dependencies (necessary for integration) can be expressed.

Figure 2.1 illustrates a simple example how composite dialog structures may yield interleaving scenarios. It represents three user scenarios in the style of activity diagrams. The first scenario ”Scrolling” consists of three elementary activities of a user, more precisely, requesting the next item (”next”), going back again (”prev”), and finally jumping to the first item (”first”) within a collection. It is a possible scenario of interaction pattern [Set-Based Navigation] which allows users to explore small item collections through navigation activities (cf. [Tid98, RSL00, HDP02, DLH02]). The second scenario [Shopping Cart] describes activities of collecting and removing items into/from (virtual) shopping carts. Its general facilities are described in [RSL00, DLH02] by an according interaction pattern as well. In many commercial information services as, for example, virtual book stores, both patterns are employed in a composite way. The third scenario ”Shopping” illustrates a possible integrated execution of the elementary user scenarios. It is characterized by an interleaved interplay of activities concurrently enabled by both patterns. Beside the aspect of concurrency, inter-pattern dependencies have to be resolved in this composition. For example, invocation of an activity ’pick’, intends to add the current item into the cart. However, for performing this activity the list position of the ”Scrolling” scenario must be known to the cart pattern. In this case, the composition has to resolve dependencies, for example, by exchanging contexts.

In this chapter, we introduce interaction nets as a formal model to specify recurring interaction structures. They can be understood as realizations (or instantiations) of

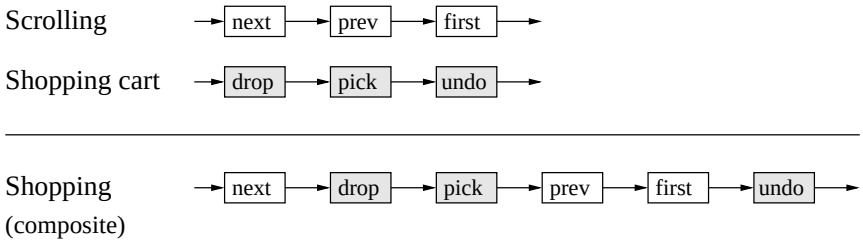


Figure 2.1: Example of an interleaved execution of two user scenario

interaction patterns. To enable concurrent composition, we base on the assumption that dialog structures can be decomposed into atomic interactive steps (or elementary activities). In the example above, activities ”next”, ”first”, etc. represent such interactive steps. Thus, an interactive component that realizes a specific interaction pattern, firstly consumes events (commonly initiated by a user), secondly processes them (for example, by adapting its internal context), and thirdly generates according actions (for example, output to the user).

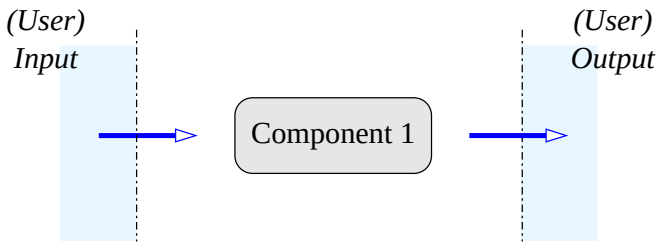


Figure 2.2: Abstract execution model of interactive components

This execution model is illustrated in Figure 2.2 whereby the represented component may be elementary as well as composite. Events initiated at an input channel are consumed by a component that reacts on this event. After processing, output is generated onto output channels. Figure 2.3 indicates the interaction between components which is enabled through composition.

2.1 Introduction to Coloured Petri Nets

We base interaction nets on the model of coloured petri nets (CP nets) which is briefly introduced in the following. For a more comprehensive introduction, we refer to an introductory book as, for example, [Jen97b]. CP nets represent a commonly applied extension to elementary place/transition nets. They basically provide a framework to

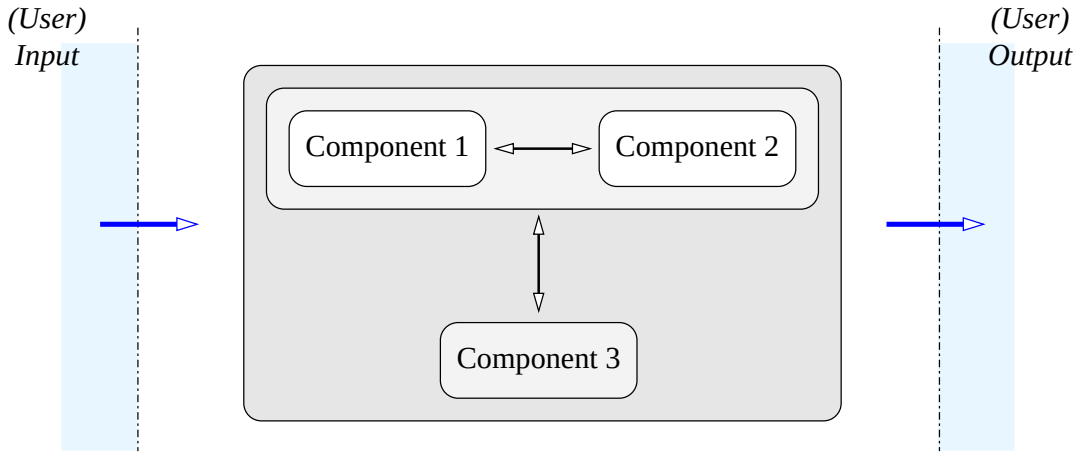


Figure 2.3: Abstract representation of interaction between components

model the flow of typed data elements and its successive processing through functions. In contrast to elementary place/transition nets, anonymous tokens are replaced by complex data elements and arc labels are extended to typed expressions.

Definition 2.1 *A* (non-hierarchical) coloured petri net (CP net) *is a tuple* $\mathcal{N} = (\Sigma, P, T, A, N, C, G, E, I)$ *which satisfies the following requirements:*

(i) Σ *is a finite set of non-empty data types, called* color sets.

(ii) $P \subset \mathbf{P}$ *is a finite set of places.*

(iii) $T \subset \mathbf{T}$ *is a finite set of transitions.*

(iv) $A \subset \mathbf{A}$ *is a finite set of arcs.*

(v) $N : A \rightarrow P \times T \cup T \times P$ *is a* node *function that associates arcs with pairs of nodes.*

(vi) $C : P \rightarrow \Sigma$ *is a* color *function that associates places with data types.*

(vii) $G : T \rightarrow EXP$ *is a* guard *function that associates transitions with expressions such that:*

$$\forall t \in T.\ type(G(t)) = bool \wedge type(var(G(t))) \subseteq \Sigma,$$

where $type(e)$ *denotes the data type of an expression* e, $type(\{e_1, e_2, \ldots\})$ *denotes the set of data types of expressions* $e_1, e_2, \ldots$, $var(e)$ *denotes the set of free variables of an expression* e, *and* EXP *denotes the set of all expression.*

(viii) $E : A \rightarrow EXP$ *is an* arc expression *function that associates arcs with expressions such that:*

$$\forall a \in A.\ type(E(a)) = C(p(a))_{MS} \wedge type(var(E(a))) \subseteq \Sigma,$$

where $p(a)$ *is the place of* $N(a)$*, and '*t_{MS}*' denotes type 'multi-set of type t'.*

(ix) $I : P \rightarrow EXP$ *is an* initialization *function that associates places with closed expressions such that:*

$$\forall p \in P.\ type(I(p)) = C(p)_{MS}.$$

Thereby, **P**, **T**, and **A** denote countable infinite sets of places, transitions, and arcs respectively, such that $\mathbf{P} \cap \mathbf{T} = \mathbf{P} \cap \mathbf{A} = \mathbf{T} \cap \mathbf{A} = \{\}$. In contrast to elementary p/t nets, an empty net structure is permitted.

Node function N maps each arc into a pair of nodes where the first element represents the source node and the second the destination node.

Color function C defines the data type of places. Thereby, a place p may contain a multi-set of data elements of type $C(p)$ only. The type system builds up on base types like *int* (integers), *real*, *string*, *bool*, and *unit* (denoting a single color represented as '()'), and enables to define new types by type constructors *subset*, *product* (tuple constructor), *record* (named tuple constructor), *union*, and *list*.

Expressions basically correspond to a variant of typed lambda calculus. The language used for CP nets is CPN ML which is an adapted version of Standard ML (SML) [Jen97b]. Thus, the evaluation of an expression is defined by means of reducing the respective expression.

Guards $G(t)$ provide an additional opportunity to control the firing of transitions t. If the guard expression evaluates to 'false', the corresponding transition must not fire. Commonly, missing guard expressions are considered as the closed expression 'true'.

Arc expression function E associates each arc with an expression. It controls which multi-sets of data elements are consumed or produced by transitions.

Initialization function I corresponds to the initial marking of classical p/t nets. It associates places with expressions that represent accordingly typed data elements.

Figure 2.4 represents an exemplary CP net. The dashed box contains data type definitions (i.e. the color sets) and assignments of data types to variables. Initialization expressions of places are distinguished by underline. Multi-sets are represented by

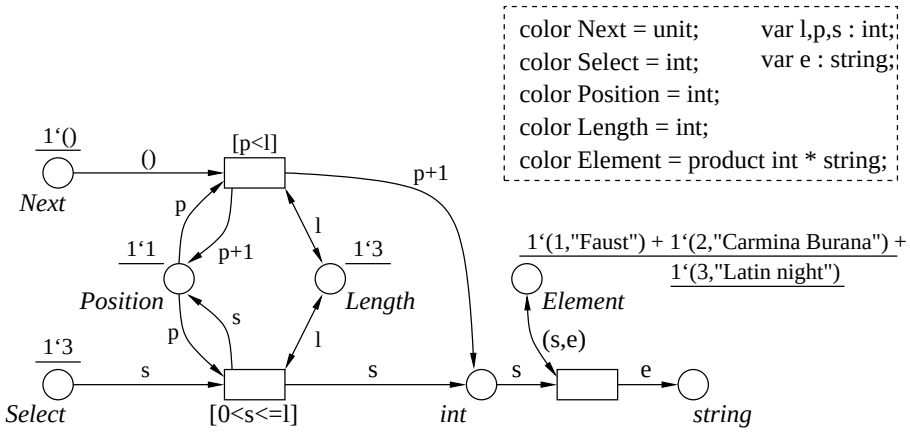


Figure 2.4: A CP net which provides basic functionality for accessing lists

’+’-separated lists of pairs of multiplicity and according data element. For example, ’1‘$4+3$‘23’ denotes multi-set $\{|4,23,23,23|\}$. As a shorthand, we allow initialization expressions and arc expressions to evaluate to a single data element ’*e*’ (not a multi-set), and interpret them as multi-set ’1‘*e*’. This shorthand is used in Figure 2.4 by arc expression ’$()$’ attached to place *Next*.

The represented net realizes a simplified access to lists. Places ’Position’, ’Length’, and ’Element’ are initialized by the start position, the number of list elements, and the list elements themselves. Initial elements ’1‘$()$’ and ’1‘3’ on places *Next* and *Select* may indicate that two events were initiated (for example, by a user). If a ’Next’ event is initiated, the current list position is incremented (if possible), and the corresponding list element is provided at the output place. In the case of an ’Select’ event, the list element at the selected position (if existing) is provided.

Net dynamics concerns the flow of data elements. Transitions generate, remove, or transform data elements. Net dynamics bases on steps which may transform the current net state into a new state. Thereby, the state of a net is defined by its marking. A state transition consists of 3 phases: (1) an *activation phase* where a set of enabled steps is determined, (2) a *selection phase* where a single step is selected for execution, and (3) an *execution phase* where the selected step occurs.

(1) Activation phase:

A set of *enabled steps* is determined.

Definition 2.2 *A* binding *b of a transition t is a function that associates all variables $v \in var(t)$ with data elements in $type(v)$ which guarantees that the guard condition $G(t) < b >$ is true. Thereby,*

G(t) $< \mathbf{b} >$ *denotes an evaluation function which computes the guard expression* $G(t)$ *by substituting its free variables* $var(G(t))$ *corresponding to binding* b *— where binding* b *is interpreted as a variable substitution,*

var(t) *denotes the set of free variables that occur in guard expression* $G(t)$ *or in an arc expression* $E(a)$ *of an arc* a *attached to transition* t*, i.e.,*

$$\forall t \in T.\ \ var(t) := \{v | v \in var(G(t)) \vee \exists a \in A(t).\, v \in var(E(a))\},$$

where $A(t)$ *denotes the set of arcs attached to* t*.*

A binding element $be = (t,b)$ *is a pair of a transition* t *and a binding* b*.*

A step Y *is a multi-set of binding elements.*

Definition 2.3 *A* token element (p,c) *is a pair of a place* p *and a data element* $c \in C(p)$*. The set of all token elements is denoted by* TE*.*

A marking m *is a multi-set over* TE*. The* initial marking m_0 *is the marking obtained by evaluating the initialization expressions, i.e.,*

$$\forall p \in P.\ \ m_0(p) := I(p) <> \ .$$

We denote the multi-set of data elements associated with a place p *in a marking* m *by* $m(p)$*.*

Note that there is a slight difference between $I(p)$ and $I(p) <>$. While $I(p)$ denotes a closed expression, $I(p) <>$ denotes the evaluation of this expression, i.e., a multi-set.

Definition 2.4 *A step* Y *is* enabled *in marking* m*, iff the following condition is satisfied:*

$$\forall p \in P.\ \ \sum_{(t,b) \in Y} E((p,t)) < b > \ \leq \ \ m(p)\,, \tag{2.1}$$

where the summation symbol denotes the union operation on multi-sets.

(2) Non-deterministic selection phase:

An enabled step Y is selected (non-deterministically) from the set computed in phase (1). The selection of an enabled step Y can be understood as:

(i) selecting a set of transitions which will be executed, and

(ii) selecting bindings wrt. each transition, i.e., selecting particular data elements which will be consumed from attached places.

(3) Execution phase ('firing'):

The execution (or *occurrence*) of an enabled step Y transforms the marking m_1 of the net to a marking m_2. Roughly, each binding element $(t,b) \in Y$ yields a transformation of marking m_1. More precisely, data elements determined by arc expressions and binding b are consumed from/produced into corresponding places attached to transition t.

Definition 2.5 *When a step Y is enabled in a marking m_1, it may* occur. *The* occurrence of Y in m_1 *transforms marking m_1 into marking m_2 by*

$$\forall p \in P.\ m_2(p) := \left(m_1(p) - \sum_{(t,b)\in Y} E((p,t)) < b > \right) + \sum_{(t,b)\in Y} E((t,p)) < b > .$$

We say that m_2 is directly reachable *from m_1 by the occurrence of step Y, denoted by: $m_1[Y\rangle m_2$.*

The sequence in which transitions are executed does not affect the resulting net marking m_2. Thus, we can think of a parallel execution.

The definition considers the dynamic semantics of the occurrence of a single step only. We extend this definition to occurrence sequences.

Definition 2.6 *An* occurrence sequence *is a finite or infinite sequence of markings and steps:*

$$m[Y_1\rangle m_1[Y_2\rangle m_2[Y_3\rangle m_3 \ldots ,$$

such that $m[Y_1\rangle m_1$, $m_i[Y_{i+1}\rangle m_{i+1}$ for all $1 \leq i < n+1$ assumed that $n \in \mathbb{N} \cup \{\infty\}$ represents the number of steps within the sequence.

We call the corresponding sequence of steps $r = \langle Y_1, Y_2, \ldots \rangle$ a run, *if the occurrence sequence starts with initial marking m_0.*

Thereby, we assume the common arithmetic laws on $\mathbb{N} \cup \infty$. In particular, $\infty + k = \infty$ and $\infty - k = \infty$ for each constant $k \in \mathbb{N}$.

As an example, we reconsider the net in Figure 2.4. By t_{next}, t_{sel}, and t_{elem}, we denote the transitions connected to places *Next*, *Select*, and *Element* respectively. Then, the following sequences $r_1, \ldots, r_6$ represent runs of the net:

$$\begin{aligned}
r_1 &= \langle \{|b_{next}|\} \rangle, \\
r_2 &= \langle \{|b_{next}|\}, \{|b_{elem2}|\}, \{|b_{sel}|\}, \{|b_{elem3}|\} \rangle, \\
r_3 &= \langle \{|b_{next}|\}, \{|b_{sel}|\}, \{|b_{elem2}|\}, \{|b_{elem3}|\} \rangle, \\
r_4 &= \langle \{|b_{next}|\}, \{|b_{sel}|\}, \{|b_{elem3}|\}, \{|b_{elem2}|\} \rangle, \\
r_5 &= \langle \{|b_{next}|\}, \{|b_{sel}|\}, \{|b_{elem2}, b_{elem3}|\} \rangle, \\
r_6 &= \langle \{|b_{sel}|\}, \{|b_{elem3}|\} \rangle,
\end{aligned}$$

if we assume the following binding elements:

$$\begin{array}{lcl} b_{next} & = & (t_{next}, \{(p,1),(l,3)\}), \\ b_{sel} & = & (t_{sel}, \{(s,3),(p,1),(l,3)\}), \\ b_{elem2} & = & (t_{elem}, \{(s,2),(e,''CarminaBurana'')\}), \\ b_{elem3} & = & (t_{elem}, \{(s,3),(e,''LatinNight'')\}). \end{array}$$

Note that runs do not necessarily reach a final marking where no further steps are enabled. For example, steps $\{|b_{elem2}|\}$ and $\{|b_{sel}|\}$ are enabled at the final marking of run r_1.

2.2 Definition of Interaction Nets

Since we intend to compose nets, we extend net specifications by an interface. The interface consists of places which are used to receive data from or provide data to externally connected components.

Definition 2.7 *An* interaction net $\mathcal{N}^i = (\mathcal{N}, P^i, P^o)$ *is defined by a CP net* $\mathcal{N} = (\Sigma, P, T, A, N, C, G, E, I)$ *and two disjunctive place sets* $P^i, P^o \subseteq P$*, assumed that* $\mathcal{N}$ *satisfies the following conditions:*

$$\forall a \in A, p \in P, t \in T. \quad (t,p) \in N(a) \Rightarrow p \notin P^i, \tag{2.2}$$

$$\forall a \in A, p \in P, t \in T. \quad (p,t) \in N(a) \Rightarrow p \notin P^o. \tag{2.3}$$

The union $P^i \cup P^o$ *is denoted by* P^{io}*. Elements of* P^i*,* P^o*, and* P^{io} *are called* input places*,* output places*, and* input/output places *(or* i/o places*) respectively.*

Input and output places can be accessed by the environment of the net. I.e., external processes may produce data elements onto input places and may consume data elements from output places. Thus, i/o places are used as an interface of the interaction net to its environment. Conditions (2.2) and (2.3) ensure that input places are exclusively used for receiving data and output places are exclusively used for providing data, i.e., an interaction net does neither generate data elements onto input places nor consume data elements from output places. Figure 2.5 represents an interaction net that results from extending the CP net in Figure 2.4 by an interface. There, i/o places are distinguished by dashed circles. Input places (as *Next* and *Select*) are distinguished from output places (as *Output*) by accordingly directed arcs. For readability, we applied the following abstractions: (i) We represent the initial marking by simple dots within places. Each dot might represent a single data element. (ii) We omit an explicit assignment of data types to variables. If omitted, we generally assume that data

types of variables are implied by the types of attached places (wrt. according arcs). (iii) The definition of data types (i.e. color sets) is directly incorporated into the net representation.

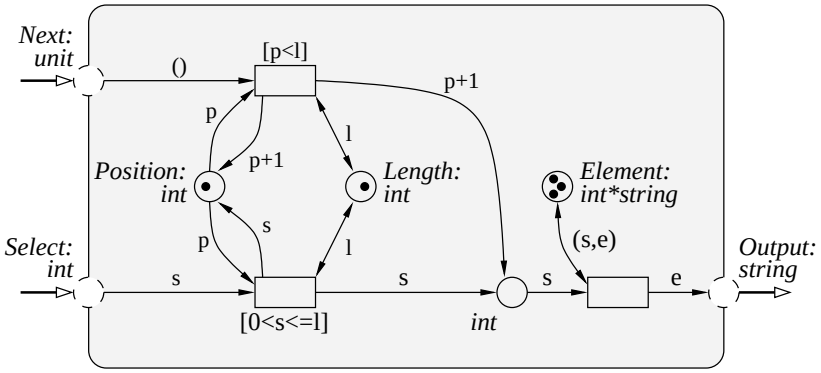


Figure 2.5: An interaction net providing event-driven navigation through lists

According to dynamic semantics, we employ the semantics of CP nets introduced above. In addition, we apply a fairness condition. For a motivation, consider CP nets in Figure 2.6. Colors for places and variables are not significant and may be, for example, *int*. (Note: we will always apply this assumption in the following, if data types are omitted.) Obviously, any run of net $\mathcal{N}_1$ contains a firing of transition t_1. However, if we consider both nets $\mathcal{N}_1$ and $\mathcal{N}_2$ as a single net $\mathcal{N}$, there exist infinite runs, where transition t_1 does never fire. These runs consist of executions of transition t_2 only. This situation generally occurs, if transitions or complete sub-nets are excluded from processing at all. Such *unfair runs* are characterized by the following definition (cf. [Rei98]):

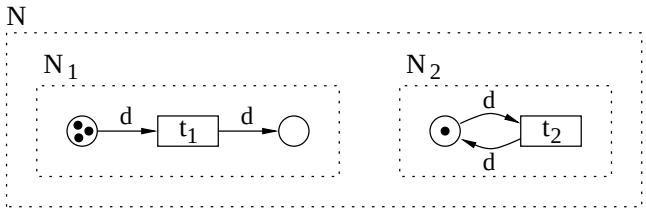


Figure 2.6: A composite CP net which permits unfair runs although its sub-nets do not

Definition 2.8 *Let $\mathcal{N}$ be a CP net, and let t be a transition of $\mathcal{N}$.*

(i) A run r of $\mathcal{N}$ neglects fairness *for t, iff t occurs only finitely often in r and is enabled infinitely often in r.*

(ii) A run r of $\mathcal{N}$ respects fairness *for t, iff r does not neglect fairness for t.*

(iii) A run r of $\mathcal{N}$ is fair, *iff r respects fairness for each transition of $\mathcal{N}$.*

Thereby, we say that a transition t is enabled in step Y, *iff there exists a binding element $(t,b) \in Y$. A transition t* occurs in step Y, *iff Y occurs, and t is enabled in Y. These conventions are extended to runs by step-wise application.*

In other words, a run is unfair, if there exist steps which are enabled for an infinite time, but are never selected for occurrence. Since situations as illustrated in Figure 2.6 may easily result from compositions, we exclude unfair runs from our considerations.

In contrast to dynamic semantics of CP nets, we permit empty steps. This relaxation provides some rather technical advantages wrt. net composition. However, it does not permit meaningless runs of empty steps as long as fairness is respected.

2.3 Utilization of Interaction Nets

At a pragmatic point of view, we distinguish input places, output places, context places, and transitions:

Input places are i/o places where external components may write into. Examples are input places which correspond to (i) events initiated by a user – for example, logging in, (ii) parameters published by another net – for example, the task history of the current session, or (iii) controlling information used for synchronization – for example, an activation signal.

Output places are i/o places where external components may read from. The interaction net conveys information via output places. If an external component connected to an output place represents a subsystem (or driver), output places represent an abstraction of external actions. For example, if an output place is externally consumed by a user interface driver or database manipulation driver, producing elements into an output place corresponds to outputting information to the user or initiating a manipulation request to a database. Depending on design and verification requirements, actions can be excluded from the interaction net by using drivers as mentioned above but can be included as well by simulating their behavior within the interaction net. For example, database management can be represented within the net by representing tables by tuple valued places and simulating database operations by transitions. Besides excluding or including drivers completely, they may be realized partially as well.

Context places are non-i/o places. They represent the internal context of the net during the execution of interactive scenarios. They can be interpreted as situation

predicates, since data elements in each context place characterize a single facet of the current situation. For example, elements in context places may represent which subtasks a user has already accomplished within an interactive task as well as preferences of the current user. If we consider the interaction net in Figure 2.5, markings of places *Position*, *Length*, and *Element* essentially determine the internal dialog situation (or context). Depending on the context, the externally observed behavior changes. For example, if the pointer reached the end of the list (i.e., $m(Position) = \{|3|\}$), *next* events will be ignored.

Transitions define context changes. In particular in user interaction, the context of interacting parties changes after each utterance [Ben98, McC93, Bun99]. To act/react properly, these changes must be reflected in the system. They are realized by transitions that compute the new context by adapting situation predicates depending on the utterance. Referring to Figure 2.5, *next* and *select* events initiate an adaptation of the list pointer.

The proposed utilization of interaction nets particularly covers a processing of events in an ECA style:

ON [**E**vent]
IF [**C**ondition]
THEN [**A**ction] .

Thereby, initiation of events is communicated via input places. Conditions are evaluated by the logic of the net. For example, *select* events outside the range are ignored in Figure 2.5. Similarly, conditions may verify access rights of the current user. Actions consist of two parts: (i) internal actions concern the context adaptation by recomputing markings of context places, and (ii) external actions concern actions not performed by the net itself, i.e., the generation of according messages which initiate subsequent processing by attached components.

Thereby, this approach enables a division of the interaction specification into atomic interactive steps (or elementary activities) which facilitates concurrent and interleaved composition. We postpone a detailed treatment, how interaction nets are utilized for the specification of interaction patterns to Chapter 4.

Dynamic interaction

The specification of ergonomic, data-intensive information services requires dynamic interaction, for example, to permit user adaptation as well as database interaction. By dynamic interaction, we understand specifications that contain conditional events whose conditions cannot be evaluated at design time or compile time. In particular, it includes user events whose activation depends on the database state which cannot

explicitly be represented at design/compile time, since database instances change over time. As interaction nets cover ECA rules, dynamic interaction may be specified as illustrated above.

2.4 Composition of Interaction Nets

In this section, we introduce a composition operator on interaction nets which is based on connecting i/o places. It requires a disjointness condition on interaction nets, such that their net structures may overlap at i/o places only. However, overlap situations can easily be resolved by renaming which will be discussed below.

In the following treatment, we assume that places possess distinguished names. To circumvent an expensive naming formalism, we apply the following convention. We consider the underlying place set **P** as a set of distinct place identifiers. Each place identifier might unambiguously be associated with a place name. Thereby, a place can equally be denoted by its identifier or its name. Transitions and arcs are treated accordingly.

For readability, we further apply a notational convention. Constituents of CP nets $\mathcal{N}$ are denoted by attaching the corresponding index. For example, places and transitions of a CP net $\mathcal{N}_1$ are denoted by P_1 and T_1, places and transitions of a CP net $\mathcal{N}'$ are denoted by P' and T'. Interaction nets are treated accordingly.

Definition 2.9 *The* composition $\mathcal{N}_1^i \circ \mathcal{N}_2^i$ *of interaction nets* $\mathcal{N}_1^i = (\mathcal{N}_1, P_1^i, P_1^o)$, $\mathcal{N}_2^i = (\mathcal{N}_2, P_2^i, P_2^o)$ *defines an interaction net* $\mathcal{N}^i = (\mathcal{N}, P^i, P^o)$ *as*

$$
\begin{aligned}
&P^i := (P_1^i \cup P_2^i) \setminus (P_1^o \cup P_2^o),\ P^o := (P_1^o \cup P_2^o) \setminus (P_1^i \cup P_2^i),\\
&\Sigma := \Sigma_1 \cup \Sigma_2,\ P := P_1 \cup P_2,\ T := T_1 \cup T_2,\ A := A_1 \cup A_2,\\
&N := N_1 \cup N_2,\ G := G_1 \cup G_2,\ E := E_1 \cup E_2,\\
&C := C_1 \cup C_2,\ I := I_1 + I_2
\end{aligned}
$$

assumed that the following properties are satisfied

$$
\begin{aligned}
T_1 \cap T_2 \;=\; A_1 \cap A_2 \;&=\; \{\}\,, && (2.4)\\
P_1 \cap P_2 \;&=\; (P_1^i \cup P_2^i) \cap (P_1^o \cup P_2^o)\,, && (2.5)\\
\forall p.\ p \in P_1 \cap P_2 \;&\Rightarrow\; C_1(p) = C_2(p)\,. && (2.6)
\end{aligned}
$$

In the definition, we extended the union operators on sets ($\cup$) and multi-sets ($+$) to functions. They are defined as:

$$(f_1 \cup f_2)(x) \;:=\; \begin{cases} f_1(x) : & x \in dom(f_1) \setminus dom(f_2), \\ f_2(x) : & x \in dom(f_2) \setminus dom(f_1), \\ f_1(x) : & x \in dom(f_1) \cap dom(f_2) \wedge f_1(x) = f_2(x), \\ \mathit{undefined} : & \text{otherwise.} \end{cases}$$

$$(f_1 + f_2)(x) \;:=\; \begin{cases} f_1(x) : & x \in dom(f_1) \setminus dom(f_2), \\ f_2(x) : & x \in dom(f_2) \setminus dom(f_1), \\ f_1(x) + f_2(x) : & x \in dom(f_1) \cap dom(f_2), \\ \mathit{undefined} : & \text{otherwise.} \end{cases}$$

The union of place sets $P_1 \cup P_2$ implies that equally named places merge. According to possible naming conflicts, we propose a renaming operation on interaction nets which will be defined below. Condition (2.4) verifies that transitions and arcs must not be merged. Condition (2.5) verifies that connections are established between input places and output places exclusively. Finally, condition (2.6) verifies that connected i/o places possess the same data type. This condition of type compatibility can further be relaxed by a sub-type relationship or an explicit type casting. An according discussion is postponed to Section 2.5. There, we also motivate why we have chosen a rather restricted variant of net composition.

An exemplary composition of interaction nets is illustrated in Figure 2.7. Figure 2.7(a) represents three elementary interaction nets. Possible connections are indicated by dotted lines. Figures 2.7(b) and 2.7(c) demonstrate the subsequent derivation of composition $\mathcal{N}_1^i \circ (\mathcal{N}_2^i \circ \mathcal{N}_3^i)$. Thereby, Figure 2.7(b) represents the intermediate result $\mathcal{N}_2^i \circ \mathcal{N}_3^i$, and Figure 2.7(c) represents the final composite net $\mathcal{N}_1^i \circ (\mathcal{N}_2^i \circ \mathcal{N}_3^i)$. Its interface consists of a single input place p_3 and a single output place p_2. Places p_1 and p_5 are no more interface places, since they established connections.

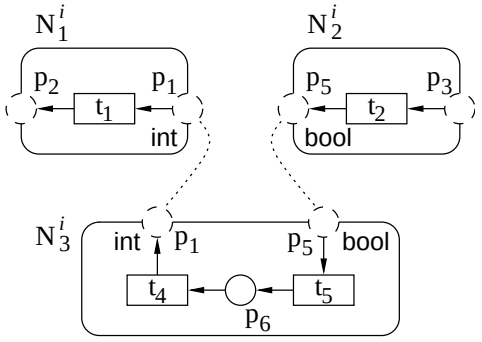
Note that Definition 2.9 is well-defined, i.e., the resulting net is in fact an interaction net. It is shown by verifying that the constituents of the composite net comply with Definitions 2.7 and 2.1.

Because of the following proposition, we can generally omit parenthesis in multiple compositions, and may neglect the order of the composition sequence. Thus, we may define $\circ\{\mathcal{N}_1^i, \ldots, \mathcal{N}_k^i\} := \mathcal{N}_1^i \circ \ldots \circ \mathcal{N}_k^i$.

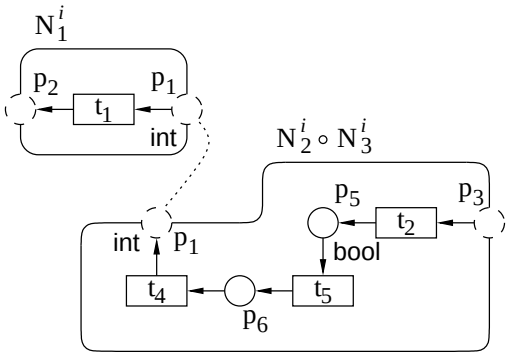
Proposition 2.1 *The composition operator $\circ$ is associative and commutative.*

Proof The proof basically reduces associativity and commutativity to that of set operators used in Definition 2.9. We first consider associativity. Let $\mathcal{N}^i := (\mathcal{N}_1^i \circ \mathcal{N}_2^i) \circ \mathcal{N}_3^i$ and $\mathcal{N}'^i := \mathcal{N}_1^i \circ (\mathcal{N}_2^i \circ \mathcal{N}_3^i)$ be two compositions. We need to show that

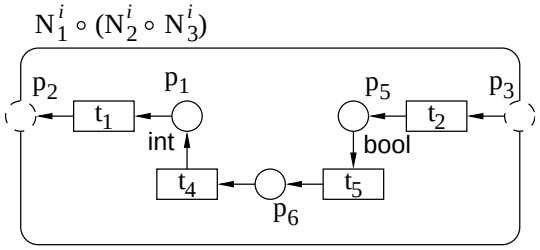
$$\mathcal{N}^i \;=\; \mathcal{N}'^i.$$



(a) Interaction nets $\mathcal{N}_1^i$, $\mathcal{N}_2^i$, and $\mathcal{N}_3^i$



(b) Net $\mathcal{N}_1^i$ and composite net $\mathcal{N}_2^i \circ \mathcal{N}_3^i$



(c) Composite net $\mathcal{N}_1^i \circ (\mathcal{N}_2^i \circ \mathcal{N}_3^i)$

Figure 2.7: Demonstration of composing interaction nets

We distinguish two cases: (1) $\mathcal{N}^i$ is defined, and (2) $\mathcal{N}^i$ is undefined.

Case (1): "$\mathcal{N}^i$ is defined"
We have to prove that (i) $\mathcal{N}'^i$ is defined as well, and (ii) both nets are equal. According to (i), we need to verify the implication: if Conditions (2.4), (2.5), and (2.6) of Definition 2.9 are valid for $\mathcal{N}^i$, then they are valid for $\mathcal{N}'^i$. According to the definition of $\mathcal{N}^i$ and $\mathcal{N}'^i$, these implications read as follows:

$$\begin{aligned}&\big[T_1 \cap T_2 = A_1 \cap A_2 = \{\} \ \wedge \ (T_1 \cup T_2) \cap T_3 = (A_1 \cup A_2) \cap A_3 = \{\}\big] \ \Rightarrow \\ &\big[T_2 \cap T_3 = A_2 \cap A_3 = \{\} \ \wedge \ T_1 \cap (T_2 \cup T_3) = A_1 \cap (A_2 \cup A_3) = \{\}\big] \ ,\end{aligned}$$

$$\begin{aligned}&\big[P_1 \cap P_2 \ = \ (P_1^i \cup P_2^i) \cap (P_1^o \cup P_2^o) \ \wedge \\ &(P_1 \cup P_2) \cap P_3 \ = \ (((P_1^i \cup P_2^i) \setminus (P_1^o \cup P_2^o)) \cup P_3^i) \cap (((P_1^o \cup P_2^o) \setminus (P_1^i \cup P_2^i)) \cup P_3^o)\big] \ \Rightarrow \\ &\big[P_2 \cap P_3 \ = \ (P_2^i \cup P_3^i) \cap (P_2^o \cup P_3^o) \ \wedge \\ &P_1 \cap (P_2 \cup P_3) \ = \ (P_1^i \cup ((P_2^i \cup P_3^i) \setminus (P_2^o \cup P_3^o))) \cap (P_1^o \cup ((P_2^o \cup P_3^o) \setminus (P_2^i \cup P_3^i)))\big] \ ,\end{aligned}$$

$$\begin{array}{l}\big[\forall p.\ \ p \in P_1 \cap P_2 \ \ \Rightarrow\ \ C_1(p) = C_2(p)\ \ \wedge \\ \ \ \forall p.\ \ p \in (P_1 \cup P_2) \cap P_3 \ \ \Rightarrow\ \ (C_1 \cup C_2)(p) = C_3(p)\big] \ \ \Rightarrow \\ \big[\forall p.\ \ p \in P_2 \cap P_3 \ \ \Rightarrow\ \ C_2(p) = C_3(p)\ \ \wedge \\ \ \ \forall p.\ \ p \in P_1 \cap (P_2 \cup P_3) \ \ \Rightarrow\ \ C_1(p) = (C_2 \cup C_3)(p)\big] \ \ .\end{array}$$

They can directly be verified by applying properties of set operators '$\cup$', '$\cap$', '$\setminus$'. We exemplarily demonstrate the proof of the first implication in detail:

$$\{\} = (T_1 \cup T_2) \cap T_3 = (T_1 \cap T_3) \cup (T_2 \cap T_3)\,,$$

because of distributivity. It implies that

$$(T_1 \cap T_3) = \{\} \ \ \wedge \ \ (T_2 \cap T_3) = \{\}\,. \tag{2.7}$$

Furthermore, since $T_1 \cap T_2 = \{\}$ is assumed by the implication, and $T_1 \cap T_3 = \{\}$ is provided by Statement 2.7, we obtain

$$\{\} = (T_1 \cap T_2) \cup (T_1 \cap T_3) = T_1 \cap (T_2 \cup T_3)\,.$$

The implication of according conditions on arc sets A_1, A_2, and A_3 is analogous.

Equivalence of nets $\mathcal{N}^i$ and $\mathcal{N}'^i$ can be confirmed by comparing their net structures according to Definition 2.9. Their place sets P and P' are equal, since

$$P \ \ = \ \ (P_1 \cup P_2) \cup P_3 \ \ = \ \ P_1 \cup (P_2 \cup P_3) \ \ = \ \ P'$$

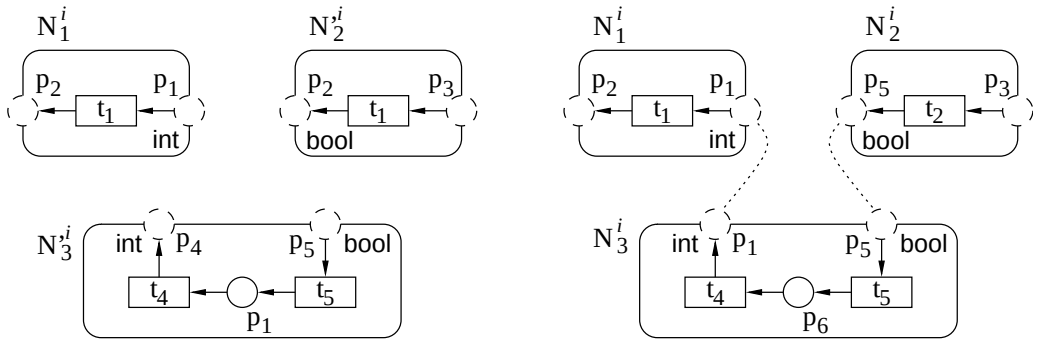
due to associativity of the union operator. The proof of equality of the color sets, transitions, etc. can be confirmed analogously by applying properties of set operators. Note that the introduced extension of operators '$\cup$' and '$+$' to functions does preserve associativity and commutativity.

Case (2): "$\mathcal{N}^i$ is undefined"
The proof of this case is analogous to part (i) of Case (1). In contrast, the three implications must be verified in the opposite direction (applying the principle of contraposition).

Commutativity is analogously proved by considering according cases (1) and (2). Its verification is accordingly based on the properties of set operators. □

Because of Conditions (2.4) – (2.6), there exist interaction nets which cannot be composed directly. As an example, consider the nets represented in Figure 2.8(a). There exist several conflicts. For example, Condition (2.4) is violated, since transition t_1 occurs in $\mathcal{N}_1^i$ and $\mathcal{N}_2'^i$. Condition (2.6) is violated as well, since place p_1 is used as inner place in $\mathcal{N}_3'^i$ and as input place in $\mathcal{N}_1^i$.

However, we may generically resolve such overlap situations by renaming places, transitions, and arcs. An according renaming operator is defined in the following. Besides conflict resolution, it can be utilized to specify desired connections. For example, the renaming applied to nets in Figure 2.8(a) establishes connections between p_1 of $\mathcal{N}_1^i$ and p_4 of $\mathcal{N}_3'^i$, and between p_2 of $\mathcal{N}_2'^i$ and p_5 of $\mathcal{N}_3'^i$ — indicated by dotted lines in Figure 2.8(b).



(a) Nets $\mathcal{N}_1^i$, $\mathcal{N}_2'^i$, $\mathcal{N}_3'^i$ before renaming

(b) After renaming:
$\mathcal{N}_2^i := \rhd_{f_2}(\mathcal{N}_2'^i)$, $\mathcal{N}_3^i := \rhd_{f_3}(\mathcal{N}_3'^i)$

Figure 2.8: Demonstration of renaming interaction net

Definition 2.10 *A* renaming $\rhd_f$ *of an interaction net* $(\mathcal{N}, P^i, P^o)$ *is defined by a total, injective function* $f : (P \cup T \cup A) \to (\mathbf{P} \cup \mathbf{T} \cup \mathbf{A})$ *with* $f(P) \subseteq \mathbf{P}$, $f(T) \subseteq \mathbf{T}$, *and* $f(A) \subseteq \mathbf{A}$. *The renamed interaction net* $(\mathcal{N}', P'^i, P'^o) := \rhd_f(\mathcal{N}, P^i, P^o)$ *is defined as*

$$
\begin{aligned}
&P'^i := f(P^i),\ P'^o := f(P^o),\\
&\Sigma' := \Sigma,\ P' := f(P),\ T' := f(T),\ A' := f(A),\\
&G' := G \circ f^{-1},\ E' := E \circ f^{-1}, C' := C \circ f^{-1},\ I' := I \circ f^{-1},\\
&N'(a') := \begin{cases} (f(n_1), f(n_2)) & : \quad N \circ f^{-1}(a') = (n_1, n_2), \\ \textit{undefined} & : \quad \textit{otherwise}. \end{cases} \qquad \textit{for all } a' \in A',
\end{aligned}
$$

where $f(M) := \bigcup_{m \in M} \{f(m)\}$, f^{-1} *denotes the inverse function of* f, *and* $f_1 \circ f_2$ *denotes the common function concatenation.*

If function f is not total on $P \cup T \cup A$, we can generally apply a completion of f as $f \cup \{m \to m \mid m \in P \cup T \cup A \wedge f(m) \textit{ is undefined}\}$. Definition 2.10 is well-defined, since renaming does not affect the structure of interaction nets. Therefore, the resulting net is in fact an interaction net.

Interaction nets $\mathcal{N}_2^i$ and $\mathcal{N}_3^i$ in Figure 2.8(b) result from renaming operations $\rhd_{f_2}(\mathcal{N}_2'^i)$ and $\rhd_{f_3}(\mathcal{N}_3'^i)$ with functions $f_2 := \{p_2 \to p_5, t_1 \to t_2\}$ and $f_3 := \{p_1 \to p_6, p_4 \to p_1\}$. Overlap situations can generally be resolved by preceding net composition with a generic renaming operation. Assumed that an interaction net possesses an unambiguous net identifier. A generic renaming then might prefix names of places, transitions, and arcs by this identifier. Afterwards, a subsequent renaming may be applied to establish desired connections.

Utilization of Composition

Composition of interaction nets enables to synthesize complex dialog structures from elementary ones. However, if we established a collection of elementary patterns by means of interaction nets, composition might require to resolve sophisticated dependencies. For example, consider a composition between two interaction nets realizing patterns 'Selectable Search Space' and 'Set-Based Navigation'. Through pattern 'Selectable Search Space', a user may alter its current view by selecting a different sub-space. Afterwards (s)he may browse this sub-space by using interactive facilities provided by pattern 'Set-Based Navigation'. However, this opportunity requires that the adapted dialog context of 'Selectable Search Space' is published to pattern 'Set-Based Navigation' — otherwise the user will not experience its intended change of view. For this purpose of resolving inter-pattern dependencies, we propose to employ specific interaction nets which we will refer to as *composition components*.

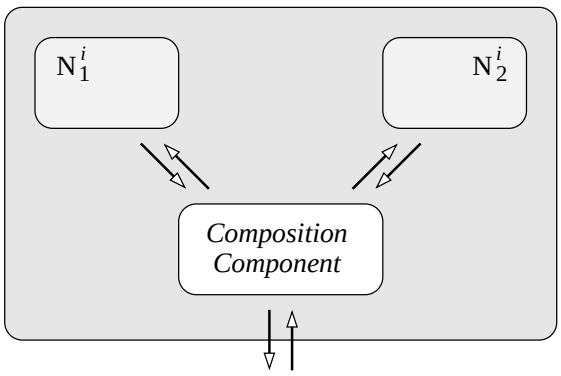


Figure 2.9: Dependency resolution by composition components

Figure 2.9 indicates how a non-elementary composition of interaction nets can be realized. There, a distinguished interaction net is employed to resolve dependencies between interaction nets $\mathcal{N}_1^i$ and $\mathcal{N}_2^i$. Although, Figure 2.9 represents an elementary composition of three interaction nets, at a semantic level, it specifies a complex composition between nets $\mathcal{N}_1^i$ and $\mathcal{N}_2^i$. Thereby, the approach scales up, since the resulting composite interaction net may in turn resolve dependencies wrt. other (possibly

complex) interaction nets. A detailed treatment of dependency resolution is postponed to Chapter 4.

2.5 Concluding Remarks

According to the composition of interaction nets, the following extensions are permissible. Firstly, the type condition of interface matching can be relaxed. Thereby, type equality can be replaced by a sub-type relationship. It requires that data types of the output places are sub-types of that of connected input places (wrt. the *subtype* constructor of CP nets). In this case, the Definition 2.9 must accordingly be rephrased, such that a merged place is associated with the data type of the former input place (i.e., the more general data type). However, we did not apply this extension, because we propose a component perspective of interaction nets in Chapter 3. It is realized by embedding the model of interaction nets into the component model introduced in [BS01]. As they assume type equality of interfaces, we currently disregard a relaxed type restriction.

However, explicit type casting generally provides the opportunity of interface matching. For this purpose, generic interaction nets are employed which compute the casting. For example, we may wish to cast type '*int*' into type '$int * int$' by function $f : int \rightarrow (int \times int)$ with $f(x) := (x, 0)$. This function is then specified by an interaction net which receives integer elements and outputs accordingly computed pairs of integers. Thus, if interaction nets $\mathcal{N}_1$ and $\mathcal{N}_2$ with type incompatible interfaces should be composed, a "mediator" net $\mathcal{M}$ is introduced to compute the desired type casting. Thereafter, the (usual) composition of all three interaction nets $\mathcal{N}_1 \circ \mathcal{M} \circ \mathcal{N}_2$ provides the desired type matching.

Another possible extension concerns multiple connections at a single interface. Although, composition of interaction nets can iteratively be applied to a set of interaction nets, composition operator '$\circ$' basically applies to pairs of argument nets only. If we drop this restriction, several interaction nets may be composed via a single i/o place. However, this opportunity might permit more compact compositions, but it does not increase expressiveness of composite nets. According n-ary connections can functionally be simulated by introducing mediator nets (as illustrated above wrt. type conversion). In addition, properties of commutativity and associativity are not applicable for the non-binary case.

In addition, there exist further extensions to composition wrt. general place/transition nets (cf., for example, [Bau96, BDK01]). For example, they permit composition at (shared) transitions as well. However, they generally do not permit a component perspective corresponding to [BS01] which is essential for our proposal.

Chapter 3

A Component Perspective

In this chapter, we propose a black-box semantics of interaction nets. In addition, we will show that this semantics together with its composition coincides with the stream-based component model introduced by Broy et. al. [BS01]. Embedding interaction nets into such a component framework yields several benefits:

- Broy et. al. established a well-founded and accepted component model. Among others, issues of (black-box) composition and component refinement are investigated. By embedding the composition model of interaction nets into the component model, we may directly adopt their results. For example, we can apply the notion of behavior and interface refinement to interaction nets, which can be used to derive a hierarchy of interaction nets. It provides an opportunity to structure an according repository which supports later reuse.

- Interfaces in general and interfaces between interaction nets in particular commonly provide a compatibility condition based on data types. They are employed to avoid meaningless interaction between connected components. For example, a component operating on string input will fail to process lists of images. However, type compatibility only covers one aspect of ”meaningful” interaction (in terms of semantic policies of composition). Besides type constraints, there often exist dynamic constraints. For example, two interacting components might exchange messages by a policy that received messages must be acknowledged to the transmitter. Although their interfaces are type-compatible, one component might ”forget” to acknowledge messages. This failure arises, for example, if a component is introduced into an environment which does not provide this policy. To additionally avoid such incompatibilities, we introduce a semantic extension to interfaces. It is based on the component framework of Broy et. al. [BS01], and employs the notion of (interface) assertions.

- We can extend the scope of composition. Thus, besides net specifications, components specified in different styles may be used for composition.

3.1 Stream-Defined Component Model

3.1.1 Syntax and Semantics

In [BS01], Broy et. al. define a component model based on streams. *Streams* are finite or infinite sequences of data elements, called *messages*. They denote the communication histories of *directed channels*.

Definition 3.1 *Let* $\mathbf{M}$ *be a set of* messages. *A* stream $s = \langle m_1, m_2, m_3, \ldots \rangle$ *is a finite or infinite sequence of messages.* $\mathbf{M}^*$, $\mathbf{M}^\infty$, *and* $\mathbf{M}^\omega := \mathbf{M}^* \cup \mathbf{M}^\infty$ *denote the set of all finite streams, the set of all infinite streams, and the set of all streams respectively.*

For example, the stream $\langle m_1, m_2, m_3, m_1 \rangle$ observed at a communication channel indicates that first m_1 was transmitted, followed by m_2 and m_3, and finally m_1 was transmitted again. Components are connected to input and output channels. Streams received on input channels are called *input streams*, streams transmitted through output channels are called *output streams* (cf. Figure 3.1). An essential aspect of the component definition is that beside its syntactic structure, the behavior of a component must be defined formally. Basically:

> *"The behavior of a component is characterized by the relationship between input streams and output streams."*

Syntactically, components are defined as follows. An according graphical abstraction is illustrated in Figure 3.1.

Definition 3.2 *An* elementary component $\mathscr{C}$ *consists of*

(i) *a name,*

(ii) *a non-empty set of* frame labels

(iii) input declarations, *i.e., a list* $\langle i_1 : I_1, \ldots, i_n : I_n \rangle$ *of declarations of input channels where a* declaration $c : T$ *is a pair of a channel name (or channel identifier)* c *and a data type* T*,*

(iv) output declarations, *i.e., a list* $\langle o_1 : O_1, \ldots, o_m : O_m \rangle$ *of declarations of output channels, and*

(v) *a* body, *i.e., a formula in predicate logic.*

Figure 3.1: Abstract representation of a component

Frame labels impose syntactic and semantic constraints onto a component. Basic frame labels are 'timed', 'untimed', and 'time synchronous' whose semantics is explained below. Input and output declarations associate a component with an interface which accepts messages of according data types. Channel names must be unique within a declaration list. The body specifies the behavior of a component. It defines the relationship between input streams and output streams. Besides the use of predicate logic, Broy et. al. propose different specification styles, for example, a style based on state transition diagrams.

To specify component behavior by logical formulas, operators and relations on streams are employed. In the thesis, we will use the following operators and relations:

length #**s***:* provides the length of stream *s*,

concatenation $\mathbf{s_1} \frown \mathbf{s_2}$*:* provides the concatenation of streams s_1, s_2,

rest **rt**.**s***:* provides stream *s* with its first message removed,

prefix relation $\mathbf{s_1} \sqsubseteq \mathbf{s_2}$*:* provides a partial order on streams defined by

$$s_1 \sqsubseteq s_2 \quad \Leftrightarrow_{\mathrm{df}} \quad \exists r.\ s_1 \frown r = s_2,$$

truncation $\mathbf{s}|_{\mathbf{n}}$*:* provides a stream which consists of the first *n* messages of stream *s*, and

filter $\mathbf{A} \circledS \mathbf{s}$*:* provides the sub-stream of *s* obtained by removing all messages in *s* that do not occur in set *A*.

Figure 3.2 graphically represents an untimed component named "Unbounded Buffer". We assume that type *D* denotes a set of data elements and type $G := D \cup \{req\}$ where *req* denotes a request message. Thus, the buffer either receives data elements, or requests '*req*' at its input channel *i*, and produces data elements at its output channel *o*. Its intended task is to transmit messages on request in the same order they were received. It can be specified by the body:

$$\mathsf{Body}: \quad o \sqsubseteq D \circledS i \ \wedge \ \#o = \#(\{req\} \circledS i)\,. \tag{3.1}$$

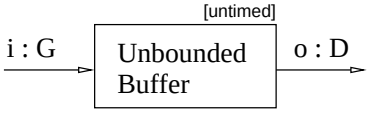


Figure 3.2: Graphical representation of an unbounded buffer

The left part of Conjunction (3.1) specifies that an output stream must be a prefix of any input stream. The right part specifies that the number of messages provided at the output exactly corresponds to the number of requests. Note that Formula (3.1) is not defined for input streams which carry more requests than actual data messages. To exclude these cases, the body must be extended by an according condition:

$$\begin{aligned} \textsf{Body:}\quad & \#(\{req\} \circledS\, i) \leq \#(D \circledS\, i) \ \wedge \\ & o \sqsubseteq D \circledS\, i \ \wedge \ \#o = \#(\{req\} \circledS\, i)\,. \end{aligned} \tag{3.2}$$

The complete specification can be represented non-graphically by the following notation:

Unbounded Buffer [untimed]
in $i : G$ out $o : D$
$\#(\{req\} \circledS\, i) \leq \#(D \circledS\, i) \ \wedge$ $o \sqsubseteq D \circledS\, i \ \wedge \ \#o = \#(\{req\} \circledS\, i)$

Dynamic semantics of components

We use the following abbreviations:

$$\begin{array}{llll} i_S := i_1,\ldots,i_n & & o_S := o_1,\ldots,o_m \\ I_S := I_1,\ldots,I_n & & O_S := O_1,\ldots,O_m \\ I_S^{\infty} := I_1^{\infty},\ldots,I_n^{\infty} & & O_S^{\infty} := O_1^{\infty},\ldots,O_m^{\infty} \\ i_S \in I_S^{\infty} := i_1 \in I_1^{\infty},\ldots,i_n \in I_n^{\infty} & & o_S \in O_S^{\infty} := o_1 \in O_1^{\infty},\ldots, \\ B_S := \mathit{Body} & & \qquad\qquad o_m \in O_m^{\infty} \end{array}$$

where M^{∞} denotes the set of infinite *timed streams* over M. Besides messages of M, timed streams may contain so-called *time ticks* '$\surd$'. A time tick denotes the end of a fixed unit of time. For example, consider the infinite timed stream

$$\langle \surd, m_1, m_2, \surd, \surd, m_3, m_1, \surd, \surd, \surd, \ldots \rangle .$$

It corresponds to a transmission of no message in the first time unit, messages m_1, m_2 in the second, no message in the third, messages m_3, m_1 in the fourth, and no messages in the following time units. Its untimed interpretation corresponds to the finite stream $\langle m_1, m_2, m_3, m_1 \rangle$.

By help of the notations introduced above, we represent the syntactic interface of a component by $i_S : I_S \rhd o_S : O_S$.

Definition 3.3 *The denotation $[\![\mathscr{C}]\!]$ of a timed elementary component $\mathscr{C}$ is defined by the formula:*

$$[\![\mathscr{C}]\!] \;:=\; i_S \in I_S^{\infty} \;\wedge\; o_S \in O_S^{\infty} \;\wedge\; B_S \,. \tag{3.3}$$

Free variables in Formula (3.3) correspond to input and output streams of component $\mathscr{C}$. This open formula determines a predicate $\mathscr{R}_S$ through

$$(i_S, o_S) \in \mathscr{R}_S \;\Leftrightarrow\; B_S \,. \tag{3.4}$$

The i/o behavior of component $\mathscr{C}$ is then specified by relation $\mathscr{R}_S$. The transition function from component specification $\mathscr{C}$ to its associated i/o behavior relation $\mathscr{R}_S$, we denote by $\mathbf{io}(\mathscr{C})$.

Besides timed specifications, components may be declared as untimed and time-synchronous. Both are special cases of the timed case. These classes differ in the interpretation of their channel identifiers in the body formula. In timed specifications, they denote timed streams of infinite length, in untimed specifications, they denote untimed streams of finite or infinite length, and in time-synchronous specifications, they denote untimed streams of infinite length. Thereby, time-synchronous streams are understood as sequences containing exactly one message at each time unit.

The semantics of untimed elementary components is reduced to the timed case. A generic transformation 'timed()' is introduced to convert an untimed elementary component specification $\mathscr{C}$ into a timed specification $\mathsf{timed}(\mathscr{C})$. Transformation $\mathsf{timed}(\mathscr{C})$ is obtained from $\mathscr{C}$ by:

1. replacing the frame label untimed by timed, and
2. substituting free variables v in the body of $\mathscr{C}$ (i.e., identifiers of input and output streams) by their "time abstraction" $\bar{v}$. The "time abstraction" operator '$\bar{}$' converts a timed stream v into an untimed stream $\bar{v}$ by removing all time ticks ($\surd$) from v:

$$\bar{v} \;:=\; M \circledS v,$$

assumed that v is declared by $v : M$.

Then, the denotation $[\![\mathscr{C}^u]\!]$ of an untimed elementary component $\mathscr{C}^u$ is defined by:

$$[\![\mathscr{C}^u]\!] \;:=\; [\![\mathsf{timed}(\mathscr{C}^u)]\!].$$

Note that although time abstraction operator $\bar{v}$ converts timed streams into their untimed representation, above transformation $\mathsf{timed}()$ in fact converts an untimed component specification into a timed specification. The essential step of the transformation is realized by step (1.) The replacement of frame label 'untimed' by label 'timed' implies a change of the data types of all variables. Thereby, variables occurring in the body formula are afterwards interpreted as timed streams. In other words, the specification then represents a relation on timed streams. Step (2.) only adjusts the body formula, such that its intention does not change within the timed perspective.

Reconsider the specification of an unbounded buffer above. It corresponds to the timed specification:

Unbounded Buffer [timed]
in $\bar{i} : G$ out $\bar{o} : D$
$\#(\{req\} \circledS \bar{i}) \leq \#(D \circledS \bar{i}) \;\wedge$ $\bar{o} \sqsubseteq D \circledS \bar{i} \;\wedge\; \#\bar{o} = \#(\{req\} \circledS \bar{i})$

It can be verified easily that pairs of input/output streams $\big(\langle 12, \mathrm{req}, 7, 11, \mathrm{req}\rangle, \langle 12, 7\rangle\big)$ and $\big(\langle 12, \mathrm{req}, \mathrm{req}, 11\rangle, \langle 12, 11\rangle\big)$ are elements of its (untimed) i/o behavior relation, while pairs $\big(\langle 12, \mathrm{req}, 7, 11, \mathrm{req}\rangle, \langle 13\rangle\big)$ and $\big(\langle 12, \mathrm{req}, 7, \mathrm{req}, \mathrm{req}\rangle, \langle 12, 7\rangle\big)$ are not. Semantics of time-synchronous specifications are reduced to the timed case by an analogous transformation (cf. [BS01]).

Note that component specification explicitly permits non-determinism. More precisely, each input stream may be associated with many possible output streams. For example, the specification of an unbounded buffer at a timed perspective might not determine an exact response time for answering requests. Thereby, an input stream of actual data and requests is related to many output streams — reflecting different response times. Thereby, non-determinism can be applied as an opportunity to model incomplete specifications and to characterize component refinement. For example, consider a specification of a mediator component which receives request from several

clients and provides them successively to a service provider. At a first design, no concrete order of occurring client requests might be specified. The according behavior relation then associates input streams from the clients with several alternative output streams to the provider — reflecting the different opportunities of ordering received requests. At a refined specification, an order might be imposed by a "first-in first-out" policy which reduces the non-determinism of the first specification. A more detailed discussion of component refinement is postponed to Section 3.3.

Component composition

The component model permits the composition of components. An essential property of component composition is that dynamic semantics of the composition is completely determined by the dynamic semantics of the elementary components.

Definition 3.4 *A* composite component $\mathscr{C} = \otimes\{\mathscr{C}_1, \ldots, \mathscr{C}_k\}$ *is defined by a name and a set of components* $\mathscr{C}_1, \ldots, \mathscr{C}_k$ *that satisfy the following condition: if* $c : T_1$ *and* $c : T_2$ *occur in channel declarations of two components* $\mathscr{C}'$ *and* $\mathscr{C}'' \in \{\mathscr{C}_1, \ldots, \mathscr{C}_k\}$ *respectively, then their data types* T_1 *and* T_2 *are equal.*

Therefore, composition requires type compatibility of interfaces. The composition establishes connections between equally named input/output channels. Connected channels are called *local channels*. The list of local channels is denoted by l_S, their corresponding data types by L_S. Figures 3.3 and 3.4 demonstrate two exemplary compositions. While the first composes components sequentially, the second composes them by means of feedback. The resulting composite components both provide as interface two input channels and two output channels. Generally, the interface of a composite component is unambiguously determined by the union of the interfaces of all sub-components minus all local channels (i.e., channels that establish connections).

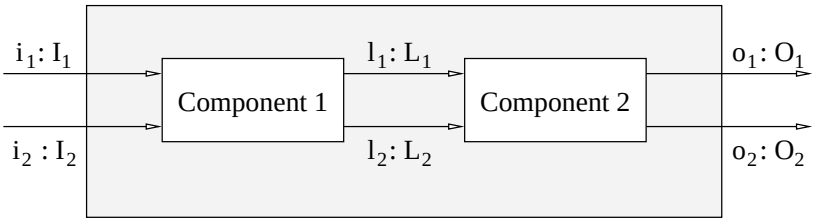


Figure 3.3: A sequential composition of two components

Definition 3.5 *The denotation* $[\![\mathscr{C}]\!]$ *of a composite component* $\mathscr{C} = \otimes\{\mathscr{C}_1, \ldots, \mathscr{C}_k\}$ *is defined by the formula:*

$$[\![\mathscr{C}]\!] \;:=\; \exists l_S \in L_S^{\infty}.\;\; \wedge_{j=1}^{k} [\![\mathscr{C}_j]\!]\,. \tag{3.5}$$

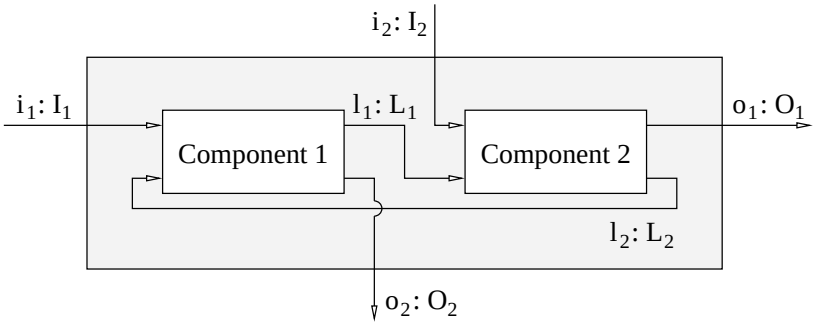


Figure 3.4: A non-sequential composition of two components

The existentially quantified channel identifiers l_S realize the desired semantics that streams generated onto an output channel of a component $\mathscr{C}'$ correspond to streams received at an input channel of a component $\mathscr{C}''$ connected to $\mathscr{C}'$. In other words, streams generated by $\mathscr{C}'$ are consumed by $\mathscr{C}''$ (wrt. a specific channel).

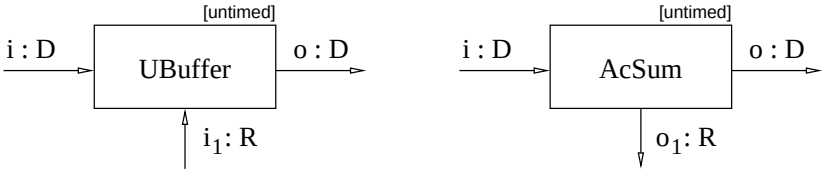
Figure 3.5 demonstrates a composition of two concrete components "UBuffer" (a slight adaptation of the buffer defined above) and "AcSum" (realizing an accumulated sum). In contrast to the buffer defined above, the single input channel is decomposed at "UBuffer" into two channels: one for receiving actual data elements and one for receiving requests. We introduce data type $R := \{req\}$ which exclusively denotes a single request message. Component "AcSum" requests numbers from the buffer and, for each received number, it computes the accumulated sum of all numbers received so far. The specification of "UBuffer" is directly derived from "Unbounded Buffer":

UBuffer	[untimed]
in $i : D, i_1 : R$ out $o : D$	
$\#i_1 \leq \#i \ \wedge \ o \sqsubseteq i \ \wedge \ \#o = \#i_1$	

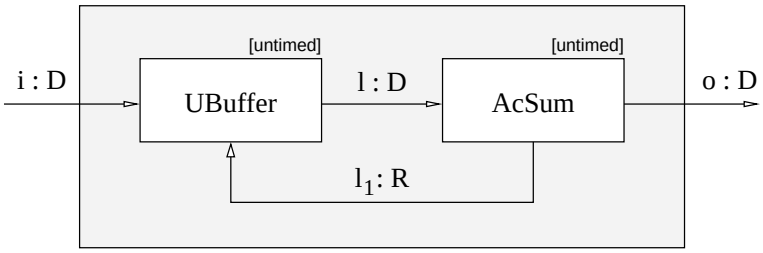
"AcSum" is specified by

AcSum	[untimed]
in $i : D$ out $o : D, o_1 : R$	
$\#i = \#o_1 \ \wedge \ \#o = \#i \ \wedge \ \left[\forall j. \ j \leq \#o \ \Rightarrow \ o.j = \sum_{k=1}^{j} i.k\right]$	

Thereby, we assume that D denotes a numeric data type. Equality $\#i = \#o_1$ requires that the number of initiated requests corresponds to the number of messages received. For the specification of the accumulated sum, we employ the term '$s.j$' to denote the j-th message of a stream s.



(a) Component "UBuffer" (b) Component "AcSum"



(c) Composition (after renaming channel identifiers)

Figure 3.5: Composition of components "UBuffer" and "AcSum"

According to the composition of "UBuffer" and "AcSum", we apply a renaming of channels as illustrated in Figure 3.5(c). The denotation of the composite component "UBuffer $\otimes$ AcSum" is derived from Definition 3.5 as

$$\begin{aligned} \exists l, l_1. \quad & \Big[\#l_1 \leq \#i \ \wedge \ l \sqsubseteq i \ \wedge \ \#l = \#l_1\Big] \ \wedge \\ & \Big[\#l = \#l_1 \ \wedge \ \#o = \#l \ \wedge \ \big[\forall j. \ j \leq \#o \ \Rightarrow \ o.j = \sum_{k=1}^{j} l.k\big]\Big] \\ \Leftrightarrow \qquad & \#o \leq \#i \ \wedge \ \big[\forall j. \ j \leq \#o \ \Rightarrow \ o.j = \sum_{k=1}^{j} i.k\big], \end{aligned} \tag{3.6}$$

where we neglected time abstraction for readability. Note that an elementary component specification with formula (3.6) as body and an according interface definition is equivalent to the composite component "UBuffer $\otimes$ AcSum". In general, elementary component specifications do not distinguish from composite component specifications. Therefore, the component approach iteratively scales up to complex components.

3.1.2 Behavioral Characterization of Composition

It is straightforward to consider i/o behavior relations specified by a component as (albeit infinite) database relations. The syntactic interface $i_S : I_S \rhd o_S : O_S$ defines an according relation schema by (i) identifying attribute names with channel names and (ii) identifying attribute domains with streams over according message types. We denote a corresponding relation schema by $S_{\mathscr{C}} = (\tilde{i}_S, \tilde{o}_S)$.

We consider the composition of two components $\mathscr{C}_1$ and $\mathscr{C}_2$. Their according relation schemes can be represented as $S_{\mathscr{C}_1} = (\tilde{i}_{S_1}, \tilde{l}_{S_1}, \tilde{o}_{S_1})$ and $S_{\mathscr{C}_2} = (\tilde{i}_{S_2}, \tilde{l}_{S_2}, \tilde{o}_{S_2})$. Thereby, $\tilde{l}_S$ denote local streams, $\tilde{i}_S$ denote (unconnected) input streams, and $\tilde{o}_S$ denote (unconnected) output streams (cf. Figures 3.3 and 3.4). By this agreement, the behavior of the composition is characterized as follows:

Proposition 3.1 *Let $\mathscr{C} = \mathscr{C}_1 \otimes \mathscr{C}_2$ be a composite component with respective i/o behavior relations $io(\mathscr{C}_1)$, $io(\mathscr{C}_2)$ and associated relation schemes $S_{\mathscr{C}_1} = (\tilde{i}_{S_1}, \tilde{l}_{S_1}, \tilde{o}_{S_1})$, $S_{\mathscr{C}_2} = (\tilde{i}_{S_2}, \tilde{l}_{S_2}, \tilde{o}_{S_2})$. Then, the composite i/o behavior relation $io(\mathscr{C})$ is determined by*

$$\begin{aligned} io(\mathscr{C}) = \{ (i_{S_1}, i_{S_2}, o_{S_1}, o_{S_2}) \mid \\ \exists l_S.\ (i_{S_1}, l_S, o_{S_1}) \in io(\mathscr{C}_1) \ \wedge \ (i_{S_2}, l_S, o_{S_2}) \in io(\mathscr{C}_2) \}. \end{aligned} \tag{3.7}$$

Proof The proposition is directly derived from Definition 3.5 and Statement (3.4). □

By applying operators of the relational algebra [Cod70], this characterization can be expressed more compactly (cf. [FT03]). Although database relations are generally finite, according operators can be applied to infinite relations as well. In particular, we may apply projection operation π and natural join operation $\bowtie$.

Proposition 3.2 *Let $\mathscr{C} = \mathscr{C}_1 \otimes \mathscr{C}_2$ be a composite component with respective i/o behavior relations $io(\mathscr{C}_1)$, $io(\mathscr{C}_2)$ and associated relation schemes $S_{\mathscr{C}_1} = (\tilde{i}_{S_1}, \tilde{l}_{S_1}, \tilde{o}_{S_1})$, $S_{\mathscr{C}_2} = (\tilde{i}_{S_2}, \tilde{l}_{S_2}, \tilde{o}_{S_2})$. Then, the composite i/o behavior relation $io(\mathscr{C})$ is determined by the natural join:*

$$\begin{aligned} \mathscr{R} &= \pi_{\tilde{i}_{S_1}, \tilde{i}_{S_2}, \tilde{o}_{S_1}, \tilde{o}_{S_2}} (io(\mathscr{C}_1) \bowtie io(\mathscr{C}_2)) \\ &= io(\mathscr{C}_1) \bar{\bowtie} io(\mathscr{C}_2), \end{aligned}$$

where $\bar{\bowtie}$ denotes the natural join with a subsequent projection onto the non-joined attributes.

Proof Statement (3.7) of Proposition 3.1 corresponds to the definition of the natural join operator. In addition, local streams l_S are projected out from the result. □

Reconsider the example at Figure 3.5. Tables 3.1 and 3.2 represent selected elements of the i/o behavior relations of components "UBuffer" and "AcSum". There, postfix '…' at the end of streams is used to denote infinite sequences of time ticks. Note that by definition, i/o behavior relations are considered in the timed perspective.

$\underline{io(\text{UBuffer})}$:

i	i_1	o
$\langle 5,11,9,\surd,\surd,\ldots\rangle$	$\langle req,req,\surd,\surd,\ldots\rangle$	$\langle 5,11,\surd,\surd,\ldots\rangle$
$\langle 5,11,9,\surd,\surd,\ldots\rangle$	$\langle req,req,\surd,\surd,\ldots\rangle$	$\langle \surd,5,11,\surd,\surd,\ldots\rangle$
$\langle 5,\surd,\surd,\ldots\rangle$	$\langle req,\surd,\surd,\ldots\rangle$	$\langle 5,\surd,\surd,\ldots\rangle$
$\ldots$	$\ldots$	$\ldots$

Table 3.1: Selected elements of behavior relation $io(\text{UBuffer})$

$\underline{io(\text{AcSum})}$:

i	o_1	o
$\langle 5,11,\surd,\surd,\ldots\rangle$	$\langle req,req,\surd,\surd,\ldots\rangle$	$\langle 5,16,\surd,\surd,\ldots\rangle$
$\langle 5,11,\surd,\surd,\ldots\rangle$	$\langle req,req,\surd,\surd,\ldots\rangle$	$\langle 5,\surd,16,\surd,\surd,\ldots\rangle$
$\langle \surd,5,11,\surd,\surd,\ldots\rangle$	$\langle req,req,\surd,\surd,\ldots\rangle$	$\langle \surd,\surd,5,16,\surd,\surd,\ldots\rangle$
$\ldots$	$\ldots$	$\ldots$

Table 3.2: Selected elements of behavior relation $io(\text{AcSum})$

$\underline{io(\text{UBuffer}\otimes\text{AcSum})}$:

i	o
$\langle 5,11,9,\surd,\surd,\ldots\rangle$	$\langle 5,16,\surd,\surd,\ldots\rangle$
$\langle 5,11,9,\surd,\surd,\ldots\rangle$	$\langle 5,\surd,16,\surd,\surd,\ldots\rangle$
$\langle 5,11,9,\surd,\surd,\ldots\rangle$	$\langle \surd,\surd,5,16,\surd,\surd,\ldots\rangle$
$\ldots$	$\ldots$

Table 3.3: Selected elements of behavior relation $io(\text{UBuffer}\otimes\text{AcSum})$

Table 3.3 represents corresponding elements of the i/o behavior relation of composition "UBuffer $\otimes$ AcSum", assumed that channel identifiers are renamed according to Figure 3.5(c). They are determined by computing the natural join (as shown by

Proposition 3.2). Thereby, the first element of $io(\text{UBuffer})$ joins with the first two elements of $io(\text{AcSum})$, the second element of $io(\text{UBuffer})$ joins with the third element of $io(\text{AcSum})$, and the third element of $io(\text{UBuffer})$ does not join (wrt. the elements selected). By applying Formula (3.6) at the timed perspective, it can be verified that the resulting elements correspond to Definition 3.5.

Note that the behavior of components is not necessarily computable completely, since specification is based on predicate logic.

As illustrated in Figure 3.6, proposition 3.2 indicates that transition function $io()$ can be understood as a homomorphism from component specifications (together with component composition $\otimes$) to behavior relations (together with behavior composition $\bowtie$), i.e.,

$$io(\mathscr{C}_1 \otimes \mathscr{C}_2) = io(\mathscr{C}_1) \bowtie io(\mathscr{C}_2)\,.$$

Strictly speaking, the transition does not represent a total homomorphism defined for any pairs of components, because the operator '$\otimes$' is partially defined on components only. For example, components with type-incompatible interfaces cannot be composed.

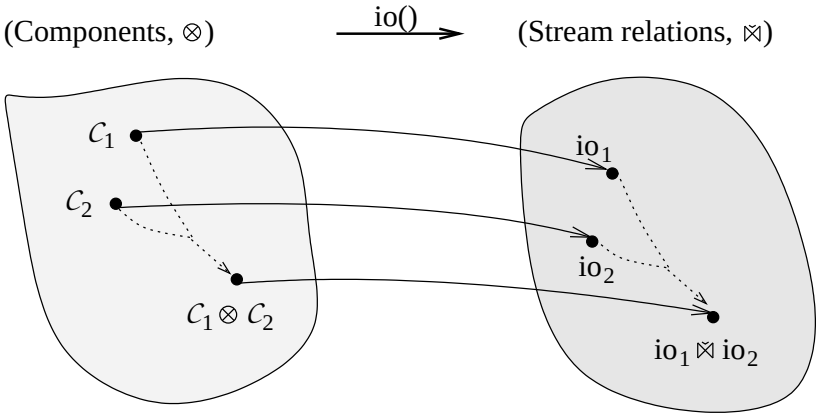


Figure 3.6: Relation between component composition and its i/o behavior

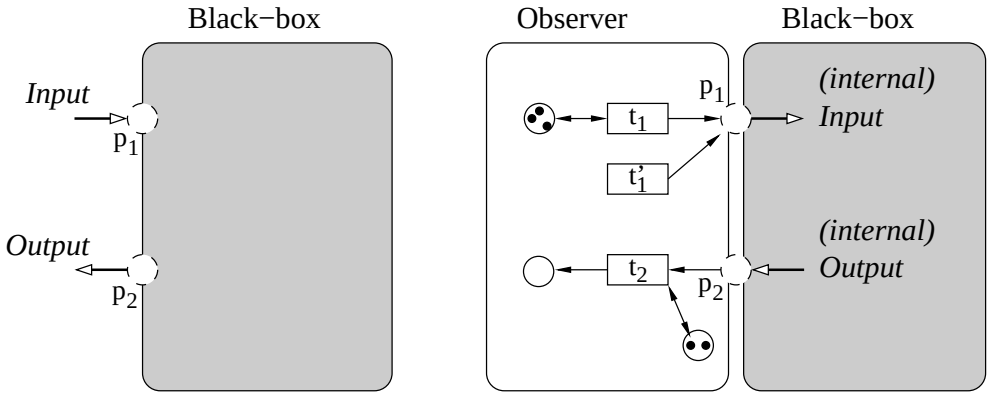
3.2 Component Semantics of Interaction Nets

In Chapter 2, we defined formal syntax and semantics of interaction nets. Thereby, dynamic semantics is specified by a net specification. In this section, we introduce a characterization of the external net behavior which is based on stream relations. In contrast to the former, it abstracts from the particular "implementation" of a net. It

provides the opportunity to view dynamics of interaction nets as black-box components. In the following treatment, we apply the restriction that all output places of interaction nets are initially empty. This rather technical restriction does not limit their expressiveness.

3.2.1 I/O Behavior

In the following, we will consider interaction nets as black-boxes. While the internal net structure is assumed to be unknown, we may obtain information about a black-box net through its interface only (cf. Figure 3.7(a)). Thereby, the *interface* of an interaction net is specified by the set P^{io} of i/o places together with their associated data types. To investigate the external behavior, we employ the notion of an observer (cf. [Bau96]). Technically, an observer is an interaction net itself which is connected to the investigated black-box net. As illustrated in Figure 3.7(b), an observer explores the behavior of a black-box by recording the relation between sent messages and received responses. As far as confusion is excluded, we will generally use $\mathcal{N}$ in the following to denote an interaction net $\mathcal{N}^i$ and its underlying CP net as well.



(a) A black-box net

(b) An observer investigating the i/o behavior of a black-box

Figure 3.7: Observing i/o behavior of interaction nets

Definition 3.6 *An interaction net $\mathcal{O}$ is called* observer of $\mathcal{N}$*, if $\mathcal{N}$ is an interaction net, and the composition $\mathcal{N} \circ \mathcal{O}$ exists. We denote the set of observers of $\mathcal{N}$ by ob$(\mathcal{N})$.*

To approach a stream-based characterization of net dynamics, we record the flow of data elements within runs. We use the intuition that tokens consumed from a place *p* by a transition *t* at an occurrence of a step *Y* "flow" from place *p* through an arc *a* to

transition t. We represent these data flows by so-called *traces*. Traces can be recorded at arcs and places:

Trace of a step wrt. an arc: comprises all data elements that flow from a place to a transition (or vice versa) at the occurrence of a step,

Input trace of a step wrt. a place: comprises all data elements that "join" a place at the occurrence of a step,

Output trace of a step wrt. a place: comprises all data elements that "leave" a place at the occurrence of a step.

If a transition does not fire at a step, then according traces correspond to the empty multi-set (abbreviated by '$\cdot$'). The concept of observed arcs can be considered as an extension of labeled transitions (cf. [Bau96]) which are used to record the firing sequence of transitions in elementary place/transition nets. We use $step(\mathcal{N})$ to denote the set of steps of net $\mathcal{N}$, and $run(\mathcal{N})$ to denote the set of runs of $\mathcal{N}$. The following definition formalizes the notion of traces according to individual steps as well as sequences of steps.

Definition 3.7 *Let $\mathcal{N} = (\Sigma, P, T, A, N, C, G, E, I)$ be a CP net.*

Trace $tr(Y)[a]$ of step Y wrt. arc *a is defined as the multi-set of tokens transition t connected to arc a produces/consumes through this arc during an occurrence of step Y, i.e.,*

$$tr(Y)[a] \;:=\; \sum_{(t,b)\in Y} E(a) < b > \,.$$

Input trace $tr(Y)[p^+]$ of step Y wrt. place *p is defined as the multi-set union of traces of step Y wrt. all arcs $a \in A$ with p as its destination node, i.e.,*

$$tr(Y)[p^+] \;:=\; \sum_{a\in A,(t,p)\in N(a)} tr(Y)[a]\,.$$

Output trace $tr(Y)[p^-]$ of step Y wrt. place *p is defined as the multi-set union of traces of step Y wrt. all arcs $a \in A$ with p as its source node, i.e.,*

$$tr(Y)[p^-] \;:=\; \sum_{a\in A,(p,t)\in N(a)} tr(Y)[a]\,.$$

Input trace $tr(r)[p^+]$ of a sequence of steps $r = \langle Y_1, Y_2, \ldots\rangle$ wrt. place *p is defined as the sequence of input traces of steps Y_k wrt. place p, i.e.,*

$$tr(r)[p^+] := \langle tr(Y_1)[p^+], tr(Y_2)[p^+], \ldots\rangle\,.$$

Output trace $tr(r)[p^-]$ of a sequence of steps $r = \langle Y_1, Y_2, \ldots \rangle$ wrt. place p *is defined as the sequence of output traces of steps* Y_k *wrt. place* p*, i.e.,*

$$tr(r)[p^-] := \langle tr(Y_1)[p^-], tr(Y_2)[p^-], \ldots \rangle .$$

Trace $tr(r)[\mathcal{N}]$ of a sequence of steps $r = \langle Y_1, Y_2, \ldots \rangle$ wrt. net $\mathcal{N}$ *is defined as the tuple of input and output traces of sequence* r *wrt. all places of* $\mathcal{N}$*, i.e.,*

$$tr(r)[\mathcal{N}] := \big(tr(r)[p_1^+], \ldots, tr(r)[p_k^+], tr(r)[p_1^-], \ldots, tr(r)[p_k^-]\big),$$
$$\text{if } P = \{p_1, p_2, \ldots, p_k\}.$$

Trace $tr(\mathcal{N})$ of net $\mathcal{N}$ *is defined as the set of traces of all runs of net* $\mathcal{N}$*, i.e.,*

$$tr(\mathcal{N}) := \bigcup_{r \in run(\mathcal{N})} tr(r)[\mathcal{N}] .$$

The definition accordingly applies to interaction nets. As introduced in Section 2.1, we employ the summation symbol to denote the union operator on multi-sets. Note that traces do not depend on actual net markings. In other words, traces may be computed according to steps which are not necessarily enabled. In particular, computing the trace of a sequence $r = \langle Y_1, Y_2, \ldots \rangle$ of steps does not require that r is a run, although we will usually be interested in traces of runs.

To illustrate the definition, we consider the interaction net in Figure 3.8. The figure represents an observer that investigates the behavior of a buffer. As net dynamics does not impose first-in/first-out behavior of data elements produced onto places, the buffer relays received messages in a non-deterministic order.

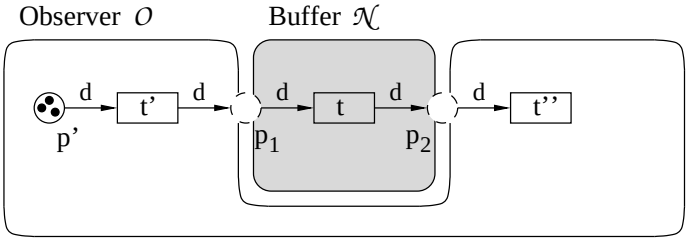


Figure 3.8: An observer connected to an unreliable buffer

Table 3.4 represents two runs r_1 and r_2 of the composite net $\mathcal{O} \circ \mathcal{N}$ assumed that place p' is initialized by multi-set $\{|1,5,8|\}$. Table 3.5 represents their corresponding traces $tr(r_1)[\mathcal{O} \circ \mathcal{N}]$ and $tr(r_2)[\mathcal{O} \circ \mathcal{N}]$. There, multi-sets are represented by the '+' notation. We omitted traces that exclusively contain empty messages, i.e., traces according to

	Y_1	Y_2	Y_3	Y_4
r_1 :	$(t',1)$	$(t',5)+(t,1)$	$(t',8)+(t,5)$	$(t,8)$
r_2 :	$(t',5)+(t',8)$	$(t',1)$	$(t,1)+(t,8)$	$(t,5)$

Table 3.4: Two runs of interaction net $\mathscr{O} \circ \mathscr{N}$

	p'^-	p_1^+	p_1^-	p_2^+
$tr(r_1)$:	$\langle 1,5,8,\cdot\rangle$	$\langle 1,5,8,\cdot\rangle$	$\langle\cdot,1,5,8\rangle$	$\langle\cdot,1,5,8\rangle$
$tr(r_2)$:	$\langle 5+8,1,\cdot,\cdot\rangle$	$\langle 5+8,1,\cdot,\cdot\rangle$	$\langle\cdot,\cdot,1+8,5\rangle$	$\langle\cdot,\cdot,1+8,5\rangle$

Table 3.5: Tabular representation of traces of runs r_1 and r_2

p'^+ and p_2^-. Since a single variable d occurs in the example net only, we dropped d from the representation of binding elements in Table 3.4.

Note that we can infer the sequence of net markings $m_1, m_2, \ldots$ from the trace of a run $r = \langle Y_1, Y_2, \ldots\rangle$, if the initial marking m_0 is known. For each place p and each step Y_k, it is inductively computed by

$$\begin{aligned} m_k(p) &= m_{k-1}(p) + tr(Y_k)[p^+] - tr(Y_k)[p^-] \\ &= m_0(p) + \sum_{j=1}^{k} tr(Y_j)[p^+] - \sum_{j=1}^{k} tr(Y_j)[p^-] \\ &= m_0(p) + \sum tr(r)[p^+] - \sum tr(r)[p^-], \end{aligned} \tag{3.8}$$

where summation on sequences of multi-sets (as used in Equality (3.8)) is defined as the multi-set union of their elements.

We adopt the notion of streams and messages from Section 3.1 for the behavior of interaction nets. We define a *(net) message* as a multi-set of data elements of the same data type. The intuition behind is that a place may receive or emit several data elements at the occurrence of a single step. Since we use a place-oriented interface, it is possible to send or receive multi-sets of data elements at a time. We apply the definitions of *streams*, *stream relations*, and *operators* on streams from Section 3.1 to (net) messages. It is straightforward to verify that

(i) the trace of a step wrt. an arc is a message,

(ii) input and output traces of a step wrt. a place are messages,

(iii) input and output traces of runs wrt. a place are streams,

(iv) a trace of an interaction net wrt. a run is an element of a stream relation, and

(v) a trace of an interaction net is a stream relation.

Therefore, we may use the notion of input/output streams as a synonym for input/output traces (of runs wrt. places). As traces of nets represent stream relations, we may adopt the database perspective from Section 3.1.2. According to attribute names of relation schemes, we use the following convention: if $P = \{p_1, p_2, \ldots, p_k\}$ is a set of places, then $(P)^+ := \{p_1^+, p_2^+, \ldots, p_k^+\}$ are attribute names that denote streams of corresponding input traces, and $(P)^- := \{p_1^-, p_2^-, \ldots, p_k^-\}$ are attribute names that denote streams of corresponding output traces. Attribute domains are inferred from data types of their corresponding places. By defining $S_{[\mathscr{N}]} := (\mathscr{N}.P)^+ \cup (\mathscr{N}.P)^-$, each (non-empty) sub-set $S_1 \subseteq S_{[\mathscr{N}]}$ represents a relation schema as, for example, schema $S_1 = \{p'^-, p_1^+, p_1^-, p_2^+\} \subseteq \{p'^+, p_1^+, p_2^+, p'^-, p_1^-, p_2^-\}$ used in Table 3.5. Note that in contrast to the external perspective applied in Section 3.1.2, $S_{[\mathscr{N}]}$ additionally includes attributes denoting internal streams. However, in accordance with Section 3.1.2, we denote the external i/o-streams of $\mathscr{N}$ by schema $S_{\mathscr{N}} := (\tilde{\imath}_S, \tilde{o}_S)$, where $\tilde{\imath}_S := (\mathscr{N}.P^i)^+$ and $\tilde{o}_S := (\mathscr{N}.P^o)^-$. We denote the internal schema by $S_{]\mathscr{N}[} := S_{[\mathscr{N}]} \setminus S_{\mathscr{N}}$. Elements of a schema S_1, we denote by x_S. The type constraint imposed on x_S is represented by '$x_S : S_1$' which reads 'x_S is of type S_1'. We adopted the sub-script '$_S$' from Section 3.1 to distinguish a tuple of streams x_S from a single stream x.

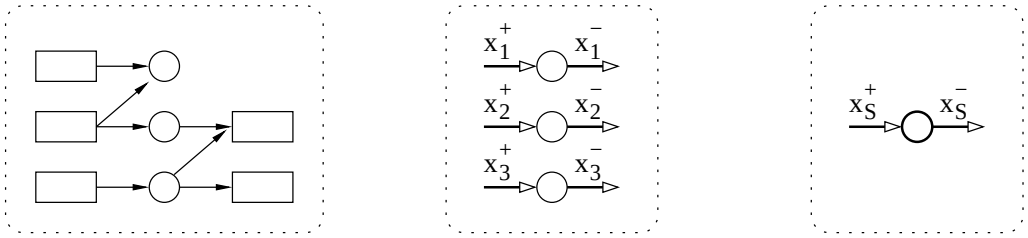
Commonly, we are not interested in the complete trace of a net $\mathscr{N}$, but in input or output traces of some places (in particular, i/o places) only. It corresponds to the projection operation of the relational algebra. For readability, we define the notation:

$$tr(\mathscr{N})[S_1] \;:=\; \pi_{S_1}(tr(\mathscr{N})),$$

if π represents the projection operator, and $S_1 \subseteq S_{[\mathscr{N}]}$. The projection operation accordingly applies to traces of runs (denoted by '$tr(r)[S_1]$') as well as to stream tuples (denoted by '$x_S[S_1]$').

A graphical abstraction that transforms an interaction net into a view that exclusively represents its streams is exemplarily illustrated in Figure 3.9. At a first abstraction step, incoming and outgoing arcs of places are subsumed to input and output streams represented by distinguished arcs (cf. Figures 3.9(a) and 3.9(b)). At a second abstraction step, input streams as well as output streams are subsumed to stream tuples represented by thick circles (cf. Figures 3.9(b) and 3.9(c)).

Figure 3.10 motivates that a single observer usually does not discover the complete i/o behavior of an interaction net. Obviously, observer $\mathscr{O}$ provides only a restricted view onto the external behavior of interaction net $\mathscr{N}$. In particular, it ignores output at place p_3 as well as possible responses to input at place p_4. In addition, input generated onto place p_1 is limited to three specific data elements. Finally, observer $\mathscr{O}$ neither tests responses to sending multi-sets of data elements, nor experiences responses of



(a) Net representation (b) Stream abstraction (c) Stream tuple abstraction

Figure 3.9: Representation of streams and stream tuples

multi-sets of data elements. Since the external behavior of a net should not depend on the particular abilities of a specific observer $\mathscr{O}$, it has to comprise input/output relations recorded by *any* observer. In the following behavior definition, we use a slightly adapted trace $tr'()$ which will be explained below.

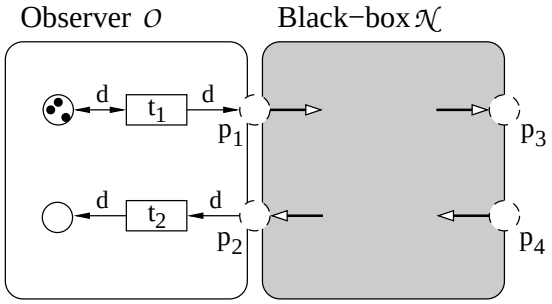


Figure 3.10: An observer that initiates input streams and records responses

Definition 3.8 *The* i/o behavior $io(\mathscr{N})$ of an interaction net $\mathscr{N}$ *is defined as the union of traces of compositions $\mathscr{N} \circ \mathscr{O}$ over all observers $\mathscr{O}$ of $\mathscr{N}$ projected to the interface streams of $\mathscr{N}$, i.e.,*

$$io(\mathscr{N}) \;:=\; \bigcup_{\mathscr{O} \in ob(\mathscr{N})} tr'(\mathscr{N} \circ \mathscr{O})[S_{\mathscr{N}}]\,.$$

Definition 3.9 *Let $\mathscr{N}$ be a CP net, let r be a sequence of steps wrt. $\mathscr{N}$, and let $p \in \mathscr{N}.P$. The* shifted *traces $tr'(r)[p^+]$ and $tr'(r)[p^-]$ are defined as*

$$\begin{aligned} tr'(r)[p^+] &:= tr(r)[p^+]|_{\#r-1}\,, \\ tr'(r)[p^-] &:= rt.tr(r)[p^-]\,, \end{aligned}$$

if #r represents the number of steps of sequence r. Shifted traces extend to schemes and CP nets by attribute-wise application.

The shifted trace $tr'(r)[\mathcal{N}]$ equals to trace $tr(r)[\mathcal{N}]$, but (i) the last message of its input streams is removed (if finite), and (ii) the first message of its output streams is removed. According to internal streams of net $\mathcal{N}$, the shifted trace obviously carries less information than the usual trace, since some messages are removed. However, if we are interested in the *external* behavior of a net, it carries the same information, because

1. the first message of an external output stream is always the empty message '$\cdot$', since we assumed that output place are initially empty, and
2. the last message of an external input stream of length l does not affect the external output streams of length l, since it cannot be processed by any transition of net $\mathcal{N}$ before step $l+1$.

As an example, we reconsider Figure 3.8. Table 3.6 represents selected elements of the behavior relation of buffer $\mathcal{N}$. Note that some of these elements are not recorded by the observer used in Figure 3.8.

$\underline{io(\text{Buffer}):}$

p_1^+ (input)	p_2^- (output)
$\langle 1,5,8,\cdot\rangle$	$\langle\cdot,1,5,8\rangle$
$\langle 1,5,8,\cdot,\cdot\rangle$	$\langle\cdot,\cdot,\cdot,1+5+8,\cdot\rangle$
$\langle 1+5+8,\cdot,\cdot,2\rangle$	$\langle\cdot,8,\cdot,1+5\rangle$

Table 3.6: Three elements of behavior relation $io(\text{Buffer})$ (wrt. Figure 3.8)

The introduction of shifted trace $tr'()$ is motivated by the composition of nets. As Figure 3.11 illustrates, it would be beneficial to know the input/output relation between i_{S_1} and $o_{S_1}^+$ instead of the relation between i_{S_1} and o_{S_1}. When composing two nets $\mathcal{N}_1$ and $\mathcal{N}_2$, *internal output* $o_{S_1}^+$ of net $\mathcal{N}_1$ corresponds to the external input stream i_{S_2} of net $\mathcal{N}_2$. Since we consider nets as black-boxes, we do not know internal streams. However, shifted trace $tr'()$ roughly represents the desired input/output relation between i_S and o_S^+. Although, the internal output $o_{S_1}^+$ is not known exactly, the shifted trace $tr'()$

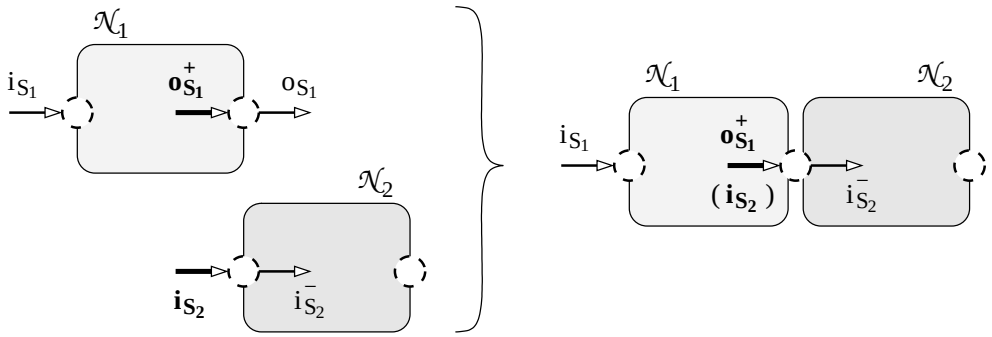


Figure 3.11: Illustration of the behavior of net composition

will suffice to permit a meaningful behavior composition. It will be derived in Section 3.2.2.

Definition 3.8 requires to investigate any observer $ob(\mathcal{N})$ to determine the i/o behavior of a net $\mathcal{N}$. Therefore, it is natural to ask if there exists a single observer that is able to reveal the complete i/o behavior of a net $\mathcal{N}$. We call such observers *general*.

Definition 3.10 *An observer $\mathcal{O}_g$ of $\mathcal{N}$ is called* general (wrt. $\mathcal{N}$)*, iff*

$$\forall \mathcal{O} \in ob(\mathcal{N}). \quad tr'(\mathcal{N} \circ \mathcal{O})[S_{\mathcal{N}}] \;\subseteq\; tr'(\mathcal{N} \circ \mathcal{O}_g)[S_{\mathcal{N}}].$$

We denote the set of general observers of $\mathcal{N}$ by $ob_g(\mathcal{N})$.

By definition, a general observer is able to simulate the behavior of any other observer. The definition directly implies that the i/o behavior of an interaction net $\mathcal{N}$ is determined by a single general observer $\mathcal{O}_g$ only (if existing), i.e.,

$$tr'(\mathcal{N} \circ \mathcal{O}_g)[S_{\mathcal{N}}] \;=\; io(\mathcal{N}). \tag{3.9}$$

Figure 3.12 motivates a generic construction of general observers. The represented interaction net $\mathcal{N}$ provides as interface an input place p_1 of some type T_1 and an output place p_2 of some type T_2. Observer $\mathcal{O}_g$ connects to each i/o place a single transition that either produces or consumes messages. Consider, for example, transition t_1. Its connected arc is labeled by a single variable x. Since x is 'unbounded', i.e., it does not occur in any incoming arc of transition t_1, it may be instantiated by an *arbitrary* data element of its domain. As the domain of x comprises multi-sets on type T_1, x may be instantiated by an arbitrary (but type-compatible) message at each step. By a corresponding argument, transition t_2 may consume any message at each step — restricted by the marking of place p_2 only.

The following algorithm generalizes this example to arbitrary interaction nets. More precisely, it constructs a general observer $\mathcal{O}_g$ according to a given interaction net $\mathcal{N}$.

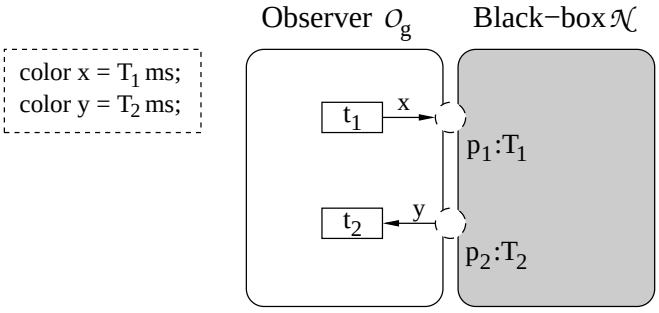


Figure 3.12: A general observer

Algorithm OBSERVER$_g(\mathcal{N})$**:**
We iteratively construct an observer $\mathcal{O}_g$ of a given interaction net $\mathcal{N}$ as follows. We start with an empty interaction net $\mathcal{O}_g$. For each i/o place $p \in \mathcal{N}.P^i \cup \mathcal{N}.P^o$:

(i) Add place p to place set $\mathcal{O}_g.P$, and adopt its data type, i.e., $\mathcal{O}_g.C(p) := \mathcal{N}.C(p)$. If $p \in \mathcal{N}.P^i$, then add p to set $\mathcal{O}_g.P^o$, otherwise add p to set $\mathcal{O}_g.P^i$.

(ii) Introduce a distinct transition t into $\mathcal{O}_g.T$. Introduce a distinct arc a into $\mathcal{O}_g.A$.

(iii) If $p \in \mathcal{N}.P^i$, then arc a directs from t to p, i.e., $\mathcal{O}_g.N(a) = (t,p)$, otherwise arc a directs from p to t, i.e., $\mathcal{O}_g.N(a) = (p,t)$.

(iv) Introduce a distinct variable x of data type $C(x) = \mathcal{O}_g.C(p)_{MS}$, and apply this variable as an arc expression of a, i.e., $\mathcal{O}_g.E(a) = x$.

The following proposition confirms that for each interaction net $\mathcal{N}$, this algorithm constructs a general observer of $\mathcal{N}$.

Proposition 3.3 *Let $\mathcal{N}$ be an interaction net. Algorithm* OBSERVER$_g(\mathcal{N})$ *computes an interaction net $\mathcal{O}_g$ which represents a general observer of $\mathcal{N}$.*

Proof By Definitions 2.1, 2.7, and 3.6, it is straightforward to verify that the algorithm computes an observer of $\mathcal{N}$. It is left to prove that observer $\mathcal{O}_g =$ OBSERVER$_g(\mathcal{N})$ is able to simulate the behavior of any other observer $\mathcal{O}$ of $\mathcal{N}$. More precisely, we need to prove the validity of implication

$$\forall i_S, o_S.\ (i_S, o_S) \in tr'(\mathcal{N} \circ \mathcal{O})[S_{\mathcal{N}}] \ \Rightarrow\ (i_S, o_S) \in tr'(\mathcal{N} \circ \mathcal{O}_g)[S_{\mathcal{N}}]\,.$$

By definition, if $(i_S, o_S) \in tr'(\mathcal{N} \circ \mathcal{O})[S_{\mathcal{N}}]$, then there exists a run r of $\mathcal{N} \circ \mathcal{O}$ such that

$$i_S = tr'(r)[\tilde{i}_S] \quad \text{and} \quad o_S = tr'(r)[\tilde{o}_S]\,.$$

By its construction, observer $\mathscr{O}_g$ is able to initiate any input streams for $\mathscr{N}$, and thus, there exists a run r_g which generates trace i_S in particular. The same argument applies for the output streams o_S, assumed that output places possess an according extent of markings. However, since there exists a run r where net $\mathscr{N}$ produced sufficient output, this condition is fulfilled. □

3.2.2 Behavioral Characterization of Net Composition

In this section, we investigate if the composition model of interaction nets coincides with the component framework introduced in Section 3.1. Therefore, we will prove that the behavioral characterization of component composition recognized in Section 3.1.2 applies to interaction nets as well. More precisely, we need to approve the question:

> *"Does transition function io() represent a (partial) homomorphism from (Interaction nets, ∘) to (Stream Relations, ⋈)?"*

The homomorphism mapping is illustrated in Figure 3.13. Analogously to component composition, the mapping does not represent a total homomorphism defined for any pairs of interaction nets, because the operator '∘' is partially defined on interaction nets only. A positive answer to this question would permit to embed the interaction net approach into the component framework introduced.

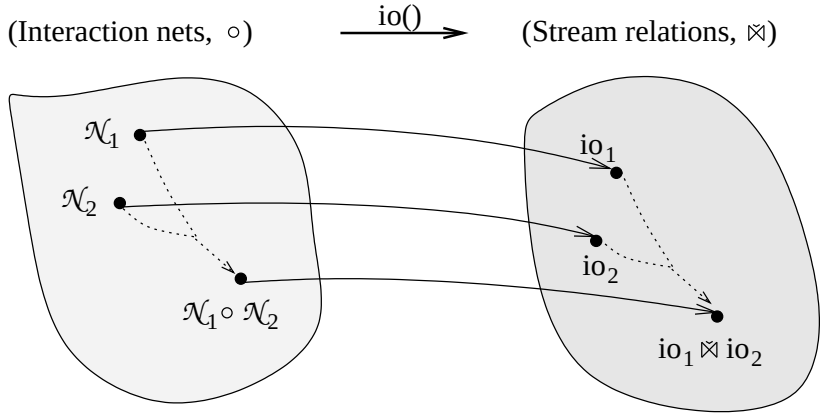


Figure 3.13: Relation between interaction net composition and its i/o behavior

To be precise, we need to comment the join operation $io(\mathscr{N}_1) \bowtie io(\mathscr{N}_2)$. Its intention is to join input streams with connected output streams. By definition 3.8, a connected pair

of input/output streams corresponds to attribute names p^+ and p^- respectively. Thus, they actually differ, and thus, the join operator would not combine them. However, as a relaxed notation we may safely omit superscripts '$+$' and '$-$' in any relation schema $io(\mathcal{N})$ without losing uniqueness. Thereby, the join computes as intended.

Before we state an according theorem, we will illustrate a motivating example. Figure 3.14 (left) represents two interaction nets $\mathcal{N}_1$ and $\mathcal{N}_2$. While $\mathcal{N}_1$ duplicates its input, $\mathcal{N}_2$ either eliminates duplicates or replaces its input by constant 'nul'. Their composition $\mathcal{N}_1 \circ \mathcal{N}_2$ represented in Figure 3.14 (right) corresponds to a unidirectional interaction from $\mathcal{N}_1$ to $\mathcal{N}_2$ realized by a single interface place.

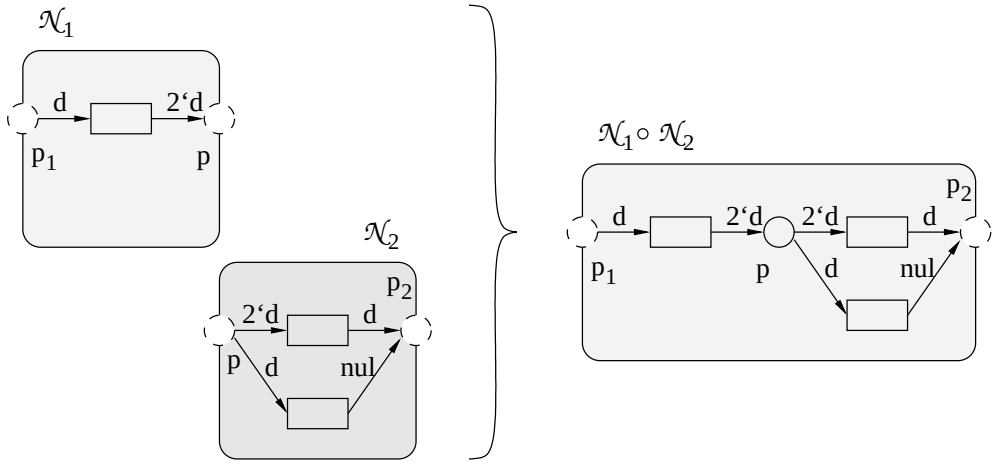


Figure 3.14: A unidirectional interaction between two interaction nets

It is straightforward to verify that Table 3.7 represents sub-sets of the i/o behavior relations $io(\mathcal{N}_1)$, $io(\mathcal{N}_2)$, and $io(\mathcal{N}_1 \circ \mathcal{N}_2)$ respectively. Consider, for example, the run $r_{1/3}$ of the composite net $\mathcal{N}_1 \circ \mathcal{N}_2$. It corresponds to a parallel execution of runs r_1 and r_3 of the elementary nets $\mathcal{N}_1$ and $\mathcal{N}_2$. In particular, output stream o of net $\mathcal{N}_1$ equals to input stream i of net $\mathcal{N}_2$. The same behavior can be observed for runs $r_{1/4}, \ldots, r_{2/7}$. These examples indicate a condition according to the behavior of the net composition: if there exist elements $(i_1, o_1) \in io(\mathcal{N}_1)$ and $(i_2, o_2) \in io(\mathcal{N}_2)$ with $i_2 = o_1$, then there exists an element $(i_1, o_2) \in io(\mathcal{N}_1 \circ \mathcal{N}_2)$. According to the example tuples, the behavior in fact corresponds to the join operation $io(\mathcal{N}_1) \bowtie io(\mathcal{N}_2)$.

Theorem 3.1 generalizes this example by considering (i) stream tuples instead of a single stream, and (ii) bidirectional interaction instead of unidirectional. It provides a positive answer to the question posed at the beginning of this section.

Theorem 3.1 *Let $\mathcal{N}_1$, $\mathcal{N}_2$ be two interaction nets. If the composition $\mathcal{N}_1 \circ \mathcal{N}_2$ exists, then the following equality is valid:*

$$io(\mathcal{N}_1 \circ \mathcal{N}_2) = io(\mathcal{N}_1) \bowtie io(\mathcal{N}_2). \tag{3.10}$$

Net	Run	Input stream	Output stream
$\mathcal{N}_1$	r_1	$\langle 1,2,3,4,\cdot\rangle$	$\langle \cdot, 1+1, 2+2, 3+3, 4+4\rangle$
	r_2	$\langle 1,2,3,4,\cdot\rangle$	$\langle \cdot, 1+1, \cdot, 3+3, 2+2\rangle$
$\mathcal{N}_2$	r_3	$\langle \cdot, 1+1, 2+2, 3+3, 4+4\rangle$	$\langle \cdot, \cdot, 1, 2, 3\rangle$
	r_4	$\langle \cdot, 1+1, 2+2, 3+3, 4+4\rangle$	$\langle \cdot, \cdot, nul, nul, nul\rangle$
	r_5	$\langle \cdot, 1+1, 2+2, 3+3, 4+4\rangle$	$\langle \cdot, \cdot, nul, 2, 3\rangle$
	r_6	$\langle \cdot, 1+1, \cdot, 3+3, 2+2\rangle$	$\langle \cdot, \cdot, nul, nul, 3\rangle$
	r_7	$\langle \cdot, 1+1, \cdot, 3+3, 2+2\rangle$	$\langle \cdot, \cdot, \cdot, 1, 3\rangle$
$\mathcal{N}_1 \circ \mathcal{N}_2$	$r_{1/3}$	$\langle 1,2,3,4,\cdot\rangle$	$\langle \cdot, \cdot, \cdot, 1, 2, 3\rangle$
	$r_{1/4}$	$\langle 1,2,3,4,\cdot\rangle$	$\langle \cdot, \cdot, nul, nul, nul\rangle$
	$r_{1/5}$	$\langle 1,2,3,4,\cdot\rangle$	$\langle \cdot, \cdot, nul, 2, 3\rangle$
	$r_{2/6}$	$\langle 1,2,3,4,\cdot\rangle$	$\langle \cdot, \cdot, nul, nul, 3\rangle$
	$r_{2/7}$	$\langle 1,2,3,4,\cdot\rangle$	$\langle \cdot, \cdot, \cdot, 1, 3\rangle$

Table 3.7: Selected elements of behavior relations $io(\mathcal{N}_1)$, $io(\mathcal{N}_2)$, $io(\mathcal{N}_1 \circ \mathcal{N}_2)$

The proof of the theorem together with required definitions and lemmas is presented in the following section. It basically verifies equality 3.10 by considering two cases:

$$\text{(L)} \quad io(\mathcal{N}_1 \circ \mathcal{N}_2) \subseteq io(\mathcal{N}_1) \bowtie io(\mathcal{N}_2)\,, \quad \text{and}$$
$$\text{(R)} \quad io(\mathcal{N}_1 \circ \mathcal{N}_2) \supseteq io(\mathcal{N}_1) \bowtie io(\mathcal{N}_2)\,.$$

Thereby, (L) can be understood as a property of decomposition and (R) as a property of composition. Roughly stated, (L) reflects the opportunity that a run observed wrt. composition $\mathcal{N}_1 \circ \mathcal{N}_2$, can be "decomposed" into two sub-runs wrt. $\mathcal{N}_1$ and $\mathcal{N}_2$. (R) reflects the opportunity that two runs observed wrt. elementary interaction nets $\mathcal{N}_1$ and $\mathcal{N}_2$ can be composed to a run wrt. their composition $\mathcal{N}_1 \circ \mathcal{N}_2$. (In addition, according observers must be taken into account.) The proof is constructive in the sense that it constructs decomposed runs from composite runs as well as composite runs from existing elementary runs.

3.2.3 Proof of the Characterization

We introduce a partial order '$\preceq$' on streams which indicates that the multi-set union of data elements of one stream is a sub-set of that of another stream for each truncation.

Definition 3.11 *The Partial order '$\preceq$' on streams is defined as:*

$$s \preceq s' \;\Leftrightarrow_{df}\; \#s \leq \#s' \;\wedge\; \left(\forall n.\; \sum_{k=1}^{n} s|_k \subseteq \sum_{k=1}^{n} s'|_k \;\vee\; n > \#s\right).$$

It extends to tuples of streams by element-wise application, i.e.,

$$(s_1,\ldots,s_k) \preceq (s'_1,\ldots,s'_k) \;\Leftrightarrow_{df}\; s_1 \preceq s'_1 \;\wedge\; \ldots \;\wedge\; s_k \preceq s'_k,$$

where the order of streams is deduced from a predefined order on their attributes.

If not stated differently, we infer a standard order on attributes from a total order on places $p \in \mathbf{P}$ together with the convention '$p^+ < p^-$'. For example, $p_1^+ < p_1^- < p_2^+ < p_2^-$. The partial order '$\preceq$' on streams can be characterized by a condition on net markings. As indicated in Figure 3.15, we consider two places p, p' with input streams i, i', and equal output streams o according to a run $r = Y_1, Y_2, \ldots$. Then,

$$\big(i \preceq i' \wedge m_0(p) \subseteq m_0(p')\big) \;\Leftrightarrow\; \forall k.\, \big(m_k(p) \subseteq m_k(p') \;\vee\; k > \#i\big). \tag{3.11}$$

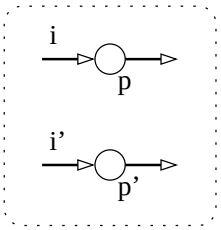


Figure 3.15: Relation '$\preceq$' on traces corresponds to relation '$\subseteq$' on markings

In other words, if $s \preceq s'$, then the marking of place p is always a subset of the marking of p', i.e., the relation $\preceq$ on streams entails the relation $\subseteq$ on place markings and vice versa.

The following lemma states necessary and sufficient conditions which a sequence of steps $r = \langle Y_1, Y_2, \ldots\rangle$ needs to satisfy to represent a run. It expresses that for each run and each place p, the input stream of p must be greater than or equal to its output stream (if we precede the input stream with the initial marking). The condition is illustrated at an example net in Figure 3.16.

Lemma 3.1 ***('Conditions of runs')*** *Let $\mathcal{N}$ be a CP net, and let $r = \langle Y_1, Y_2, \ldots\rangle$ be a sequence of steps of $\mathcal{N}$. Then, r is a run of $\mathcal{N}$, if and only if one of the following equivalent conditions are valid for all places $p \in P$:*

$$\forall k.\ k < \#r \ \Rightarrow$$

$$m_0(p) + \sum_{j=1}^{k} \sum_{(t,b)\in Y_j} E(t,p) < b > \ \supseteq \ \sum_{j=1}^{k+1} \sum_{(t,b)\in Y_j} E(p,t) < b >, \qquad (3.12)$$

$$\langle m_0(p) \rangle \frown tr(r)[p^+] \ \succeq \ tr(r)[p^-]. \qquad (3.13)$$

According to places p_e with an empty initial marking $m_0(p_e) = \{||\}$, Condition 3.13 corresponds to:

$$\langle \cdot \rangle \frown tr(r)[p_e^+] \ \succeq \ tr(r)[p_e^-], \qquad (3.14)$$

$$\langle \cdot \rangle = tr(r)[p_e^-]|_1 \ \wedge \ tr'(r)[p_e^+] \ \succeq \ tr'(r)[p_e^-]. \qquad (3.15)$$

Proof We need to prove that Conditions (3.12) – (3.15) are individually equivalent to the requirement that steps Y_1, Y_2, ... are enabled at subsequent markings of $\mathcal{N}$. Condition (3.12) states that for each place p, all data elements consumed from p until step Y_{k+1} must be produced into p before, or they must occur in the initial marking. It inductively reflects the requirement of a step to be enabled (cf. Definition 2.1). Condition (3.13) simply rewrites condition (3.12) according to Definitions 3.7 (on page 40) and 3.11 (on page 51). Conditions (3.14) and (3.15) adapt condition (3.13) to the case in which place p_e is initially unmarked. It implies that the first message of the output stream of p_e must be the empty message. □

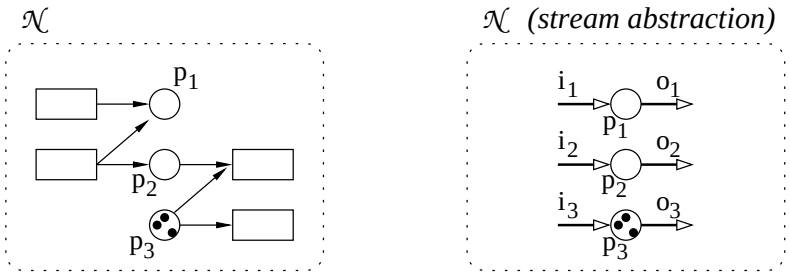


Figure 3.16: Illustration of Lemma 3.1:
$\langle \cdot \rangle \frown i_1 \succeq o_1, \ \langle \cdot \rangle \frown i_2 \succeq o_2, \ \langle m_0(p_3) \rangle \frown i_3 \succeq o_3$

To derive properties about composition and decomposition of interaction nets, we introduce operators on steps and runs. While multi-set union operator '$+$' permits composition of steps and runs, the projection operator '$[\cdot]$' permits their decomposition. Thereby, a projection of runs '$r[S]$' corresponds to determining sub-sets of binding elements which affect streams according to sub-schema S.

Definition 3.12 *Let $\mathcal{N}$ be a CP net, let $p \in P$, let $S \subseteq S_{]\mathcal{N}[}$ be a sub-schema, let $\tilde{s} \in S$ be an attribute, let Y, Y' be steps, and let $r = \langle Y_1, Y_2, \ldots \rangle$, $r' = \langle Y'_1, Y'_2, \ldots \rangle$ be sequences of steps of $\mathcal{N}$. The* composition of sequences of steps *$r + r'$ is defined as*

$$r + r' := \langle Y_1 + Y'_1, Y_2 + Y'_2, \ldots \rangle .$$

If one sequence is finite and shorter than the other, missing steps are treated as empty multi-sets. The projection of a step *$Y[\cdot]$ and the* projection of a sequence of steps *$r[\cdot]$ are defined as*

$$\begin{aligned} Y[\tilde{s}] &:= \textstyle\sum_{t \in T[\tilde{s}]} \{\!| (t,b) \in Y |\!\} , \\ Y[S] &:= \textstyle\sum_{t \in T[S]} \{\!| (t,b) \in Y |\!\} , \\ r[S] &:= \langle Y_1[S], Y_2[S], \ldots \rangle , \end{aligned}$$

where

$$\begin{aligned} T[p^+] &:= \{ t \in T \mid \exists a,t.N(a) = (t,p) \mid \} , \\ T[p^-] &:= \{ t \in T \mid \exists a,t.N(a) = (p,t) \mid \} , \\ T[S] &:= \textstyle\bigcup_{\tilde{s} \in S} T[\tilde{s}] . \end{aligned}$$

It is straightforward to verify that (i) the composition of sequences of steps $r + r'$ results in a sequence of steps, (ii) the projection $Y[S]$ of a step onto a schema results in a step, and (iii) the projection $r[S]$ of a sequence of steps onto a schema results in a sequence of steps. Note that the projection $Y[p^+]$ of a step onto attribute p^+ provides all binding elements which affect the according trace $tr(Y)[p^+]$, i.e., all binding elements of incoming transitions of place p. Accordingly, the projection $Y[p^-]$ of a step onto attribute p^- provides all binding elements of outgoing transitions of place p. Therefore, we sometimes use the notion of *affective* binding elements wrt. p^+ (or p^-). The projection intuitively extends to schemes and sequences of steps. Thereby, the definition yields the equalities $Y[S_{]\mathcal{N}[}] = Y$, and $r[S_{]\mathcal{N}[}] = r$. If $\mathcal{N} = \mathcal{N}_1 \circ \mathcal{N}_2$, we apply the abbreviations

$$Y[\mathcal{N}_k] := Y[S_{]\mathcal{N}_k[}] \quad \text{and} \quad r[\mathcal{N}_k] := r[S_{]\mathcal{N}_k[}] \quad (\text{for } k \in \{1,2\}) .$$

The following lemma states that input and output traces are exclusively determined by binding elements of accordingly projected steps.

Lemma 3.2 *Let $\mathcal{N}$ be a CP net, let $r = \langle Y_1, Y_2, \ldots \rangle$ of $\mathcal{N}$ be a sequence of steps, let $S \subseteq S_{]\mathcal{N}[}$ be a sub-schema, and let $\tilde{s} \in S$ be an attribute. Then,*

$$\begin{aligned} tr(Y_k)[\tilde{s}] &= tr(Y_k[\tilde{s}])[\tilde{s}] , \\ tr(r)[\tilde{s}] &= tr(r[\tilde{s}])[\tilde{s}] , \\ tr(r)[S] &= tr(r[S])[S] . \end{aligned}$$

Proof The equalities are directly implied from Definitions 3.7 and 3.12. □

By Definition 3.9, equalities of Lemma 3.2 are valid for shifted traces $tr'()$ as well.

Lemma 3.3 ***('Composition and decomposition of steps')*** *Let $\mathcal{N}_1$, $\mathcal{N}_2$ be interaction nets whose composition $\mathcal{N} := \mathcal{N}_1 \circ \mathcal{N}_2$ exists, and let Y, Y_1, and Y_2 be steps of $\mathcal{N}$, $\mathcal{N}_1$, and $\mathcal{N}_2$ respectively. If we apply conventions $S := S_{]\mathcal{N}[}$, $S_1 := S_{]\mathcal{N}_1[}$, and $S_2 := S_{]\mathcal{N}_2[}$, then*

$$\forall \tilde{s} \in S.\quad Y[\tilde{s}] \;=\; \begin{cases} (Y[S_1])[\tilde{s}] & : \quad \tilde{s} \in S_1\,, \\ (Y[S_2])[\tilde{s}] & : \quad \tilde{s} \in S_2\,. \end{cases} \tag{3.16}$$

$$\forall \tilde{s} \in S.\quad tr(Y)[\tilde{s}] \;=\; \begin{cases} tr(Y[S_1])[\tilde{s}] & : \quad \tilde{s} \in S_1\,, \\ tr(Y[S_2])[\tilde{s}] & : \quad \tilde{s} \in S_2\,. \end{cases} \tag{3.17}$$

$$Y[S_1] \in step(\mathcal{N}_1) \;\wedge\; Y[S_2] \in step(\mathcal{N}_2)\,, \tag{3.18}$$

$$Y_1 + Y_2 \;\in\; step(\mathcal{N})\,. \tag{3.19}$$

Statements (3.16) – (3.19) analogously apply to sequences of steps. In addition, Statement (3.17) is valid for the shifted trace $tr'()$ as well.

Proof

Statement (3.16):
It considers the projection of steps $Y[\tilde{s}]$ of a composite net $\mathcal{N} = \mathcal{N}_1 \circ \mathcal{N}_2$ onto a single attribute $\tilde{s}$. It states that if $\tilde{s}$ belongs to $\mathcal{N}_1$, then $Y[\tilde{s}]$ is completely determined by the projection of Y to $\mathcal{N}_1$ (and vice versa). In other words,

$$\begin{aligned} &\forall \tilde{s} \in S_1.\; Y[\tilde{s}] = (Y[S_1])[\tilde{s}] \;\wedge\; (Y[S_2])[\tilde{s}] = \{|\}\,, \\ &\forall \tilde{s} \in S_2.\; Y[\tilde{s}] = (Y[S_2])[\tilde{s}] \;\wedge\; (Y[S_1])[\tilde{s}] = \{|\}\,. \end{aligned} \tag{3.20}$$

It is proved by considering Definitions 3.12. The projection of a step $Y[p^+]$ onto attribute p^+ comprises all binding elements (b,t) of transitions t that produce data elements onto place p. By Definition 2.9 (on page 20), these transition either belong to sub-net $\mathcal{N}_1$ or to sub-net $\mathcal{N}_2$. Therefore, binding elements (b,t) according to p^+ occur either in projection $Y[S_1]$ or in projection $Y[S_2]$. This analogously applies to projection of step $Y[p^-]$ onto attribute p^-. Together, it implies (3.20), and thus (3.16).

Statement (3.17):
By Definition 3.7, traces of a step $tr(Y)[p^+]$ and $tr(Y)[p^-]$ are completely determined by binding elements in $Y[p^+]$ and $Y[p^-]$ respectively, i.e.,

$$\forall \tilde{s} \in S.\; tr(Y)[\tilde{s}] = tr(Y[\tilde{s}])[\tilde{s}]\,. \tag{3.21}$$

It implies that $tr(Y)[\tilde{s}] = tr(Y[S_1])[\tilde{s}]$, if $\tilde{s} \in S_1$, and $tr(Y)[\tilde{s}] = tr(Y[S_2])[\tilde{s}]$, if $\tilde{s} \in S_2$ which proves (3.17).

Statement (3.18):
Consider a binding element $(b,t) \in Y[S_1]$. By Definition 3.12, t is a transition of $\mathcal{N}_1$ and $(b,t) \in Y$. Therefore, bindings b concern arcs attached to transition t. By Definition 2.9, all arcs attached to t belong to $\mathcal{N}_1$. Therefore, all binding elements (b,t) are binding elements of $\mathcal{N}_1$. An analogous treatment of $Y[S_2]$ implies (3.18).

Statement (3.19):
We need to show that if a binding element $(b,t) \in Y_1$, then (b,t) is a binding element of $\mathcal{N}$ (and analogously for $(b,t) \in Y_2$). Since transition t is in $\mathcal{N}$, it is sufficient to realize that all arcs and places attached to t in the composite net $\mathcal{N}$ already occur in sub-net $\mathcal{N}_1$. This is ensured by the net composition operator (cf. Definition 2.9). Since guards and arc expressions are not altered by the net composition, (b,t) is a binding element of $\mathcal{N}$ as well. An analogous treatment of the case $(b,t) \in Y_2$ implies (3.19).

Statements (3.16) – (3.19) are valid for sequences of steps as well. In contrast to runs, sequences of steps do not impose further restrictions. Therefore, verification can directly be reduced to that of single steps. Note that statements (3.18) and (3.19) must be adapted such that the result of projection and composition of sequences of steps are sequences of steps as well.

By Definition 3.9, it can be verified that Statement (3.17) remains valid for shifted trace $tr'()$ as well. □

In the following, we will apply a naming convention according to stream tuples that occur in runs of composite nets $\mathcal{N}_1 \circ \mathcal{N}_2$. These names of stream tuples are illustrated at an abstract net composition in Figure 3.17 and are defined in Table 3.8.

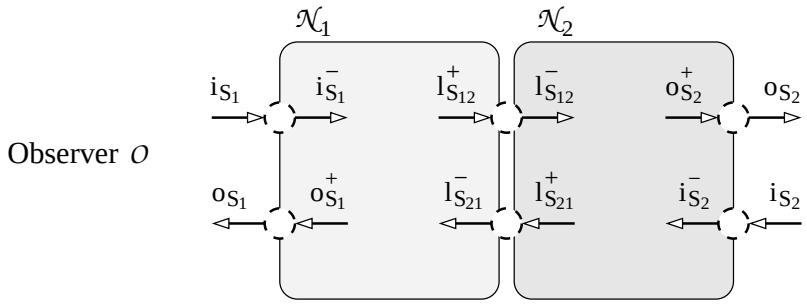


Figure 3.17: Naming convention according to streams in composite nets

The following lemma is essential for the proof of Theorem 3.1. It states that if there exist runs r_1, r_2 of two composable interaction nets $\mathcal{N}_1$, $\mathcal{N}_2$ then there exists a composite run r of $\mathcal{N}_1 \circ \mathcal{N}_2$, assumed that runs r_1, r_2 fulfill a compatibility condition. Composite run r corresponds to a parallel execution of local runs r_1, r_2. Thereby, the compatibility

Streams at $\mathcal{N}_1$	Streams at $\mathcal{N}_2$	Interface streams at $\mathcal{N}_1 \circ \mathcal{N}_2$
$i_{S_1} : (P_1^i \setminus P_2)^+$	$i_{S_2} : (P_2^i \setminus P_1)^+$	$l_{S_{12}}^+ : (P_1^o \cap P_2^i)^+$
$i_{S_1}^- : (P_1^i \setminus P_2)^-$	$i_{S_2}^- : (P_2^i \setminus P_1)^-$	$l_{S_{12}}^- : (P_1^o \cap P_2^i)^-$
$o_{S_1} : (P_1^o \setminus P_2)^-$	$o_{S_2} : (P_2^o \setminus P_1)^-$	$l_{S_{21}}^+ : (P_1^i \cap P_2^o)^+$
$o_{S_1}^+ : (P_1^o \setminus P_2)^+$	$o_{S_2}^+ : (P_2^o \setminus P_1)^+$	$l_{S_{21}}^- : (P_1^i \cap P_2^o)^-$

Table 3.8: Naming convention according to streams in $\mathcal{N}_1 \circ \mathcal{N}_2$

condition ensures that net $\mathcal{N}_1$ produces 'enough' input for net $\mathcal{N}_2$ and vice versa. The implication of Lemma 3.4 is graphically illustrated in Figure 3.18.

Lemma 3.4 ***('Composition of runs')*** *Let $\mathcal{N}_1$, $\mathcal{N}_2$ be interaction nets, Let $\mathcal{O}_1$ and $\mathcal{O}_2$ be observers of $\mathcal{N}_1$ and $\mathcal{N}_2$ respectively with $\mathcal{N}_1.P^{io} \cap \mathcal{O}_1.P^{io} = \mathcal{N}_2.P^{io} \cap \mathcal{O}_2.P^{io}$, and let r_1 and r_2 be runs of $\mathcal{N}_1 \circ \mathcal{O}_1$ and $\mathcal{N}_2 \circ \mathcal{O}_2$ respectively. If composition $\mathcal{N}_1 \circ \mathcal{N}_2$ exists, then*

$$\begin{aligned} tr(r_1)[\tilde{l}_{S_{12}}^+] = tr(r_2)[\tilde{l}_{S_{12}}^+] \ \wedge \ tr(r_1)[\tilde{l}_{S_{21}}^+] = tr(r_2)[\tilde{l}_{S_{21}}^+] \ \Rightarrow \\ (r_1[\mathcal{N}_1] + r_2[\mathcal{N}_2]) \in run(\mathcal{N}_1 \circ \mathcal{N}_2)\,, \end{aligned} \tag{3.22}$$

where $\tilde{l}_{S_{12}}^+ = (P_1^o \cap P_2^i)^+$ and $\tilde{l}_{S_{21}}^+ = (P_1^i \cap P_2^o)^+$.

Proof We define $\mathcal{N} := \mathcal{N}_1 \circ \mathcal{N}_2$, $S_1 := S_{]\mathcal{N}_1[}$, $S_2 := S_{]\mathcal{N}_2[}$, and $r := r_1[\mathcal{N}_1] + r_2[\mathcal{N}_2] = r_1[S_1] + r_2[S_2]$. First, we show that r represents a sequence of steps of $\mathcal{N}$. Therefore, we apply Lemma 3.3 (3.18) to composite net $\mathcal{N}_1 \circ \mathcal{O}_1$. It implies that $r_1[S_1]$ represents a sequence of steps of $\mathcal{N}_1$. Analogously, $r_2[S_2]$ represents a sequence of steps of $\mathcal{N}_2$. Applying Lemma 3.3 (3.19) to composite net $\mathcal{N}_1 \circ \mathcal{N}_2$ implies that $r = r_1[S_1] + r_2[S_2]$ represents a sequence of steps of $\mathcal{N}$.

To verify that r is a run of net $\mathcal{N}$, we will apply Lemma 3.1. Therefore, we have to confirm the validity of one of the conditions of Lemma 3.1 for each place $p \in P$. We distinguish three cases: (1) non-i/o places of $\mathcal{N}_1$ and $\mathcal{N}_2$, (2) i/o places of $\mathcal{N}$, and (3) i/o places of $\mathcal{N}_1$ and $\mathcal{N}_2$ which established connections.

Case 1: $p \in P_1 \setminus P_1^{io} \ \vee \ p \in P_2 \setminus P_2^{io}$:
We consider $p \in P_1 \setminus P_1^{io}$. (The other case is symmetric.) Since both attributes p^+ and p^- denote streams within $\mathcal{N}_1$, i.e., $p^+, p^- \in S_1$, we can infer

$$tr(r)[p^+] = tr(r_1)[p^+] \quad \wedge \quad tr(r)[p^-] = tr(r_1)[p^-] \tag{3.23}$$

by applying Lemma 3.3 (3.17). Since r_1 is a run of $\mathcal{N}_1$, Condition (3.13) of Lemma 3.1 is fulfilled for place p, i.e., $\langle m_0(p) \rangle \frown tr(r_1)[p^+] \succeq tr(r_1)[p^-]$. Because of Equalities (3.23), condition $\langle m_0(p) \rangle \frown tr(r)[p^+] \succeq tr(r)[p^-]$ is fulfilled for sequence r as well.

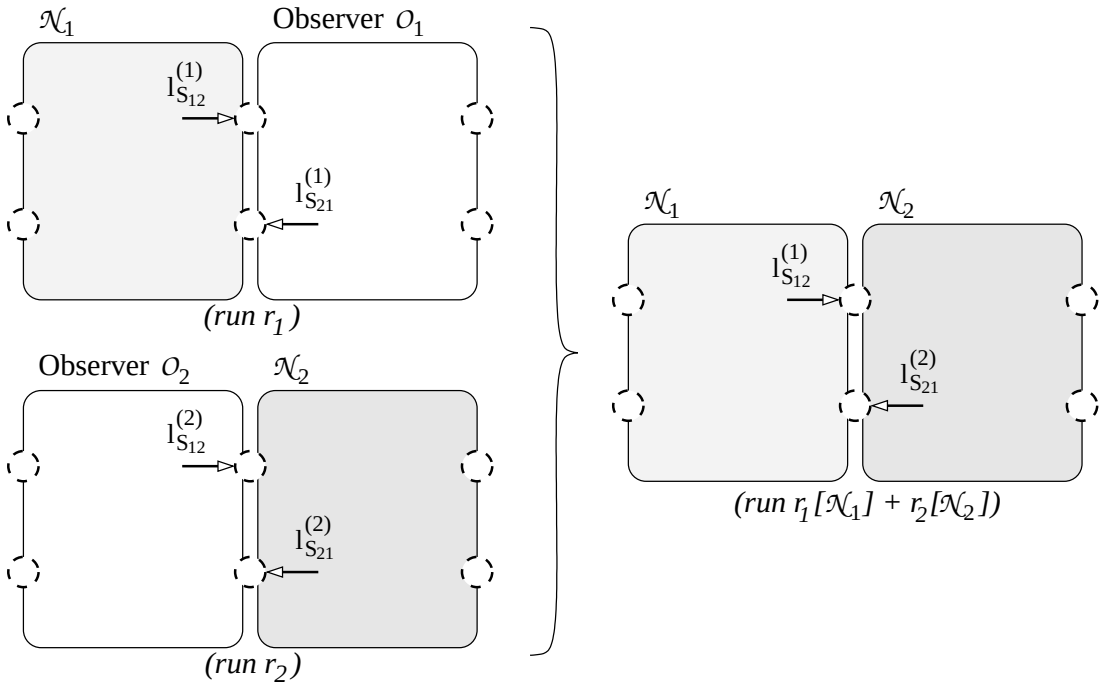


Figure 3.18: Graphical representation of Lemma 3.4:
$r_1[\mathcal{N}_1] + r_2[\mathcal{N}_2]$ is a run of $\mathcal{N}_1 \circ \mathcal{N}_2$, if $l^{(1)}_{S_{12}} = l^{(2)}_{S_{12}} \wedge l^{(1)}_{S_{21}} = l^{(2)}_{S_{21}}$.

Case 2: $p \in P^{io}$:
We first consider $p \in P^i$. By Definition 2.9, then either $p \in P^i_1$ or $p \in P^i_2$. We assume that $p \in P^i_1$. (The other case is symmetric.) Since p^- is an attribute within $\mathcal{N}_1$, i.e., $p^- \in S_1$, we can infer

$$tr(r)[p^-] = tr(r_1)[p^-] \tag{3.24}$$

by applying Lemma 3.3 (3.17). Because of assumption $\mathcal{N}_1.P^{io} \cap \mathcal{O}_1.P^{io} = \mathcal{N}_2.P^{io} \cap \mathcal{O}_2.P^{io}$ of Lemma 3.4, input place p does not possess any incoming arcs in $\mathcal{N}_1 \circ \mathcal{O}_1$. Thus, $tr(r_1)[p^+] = \langle \cdot, \cdot, \ldots \rangle$ corresponds to the trivial sequence of empty messages. Since neither $p^+ \in S_1$ nor $p^+ \in S_2$, $tr(r)[p^+] = tr(r_1[S_1] + r_2[S_2])[p^+] = \langle \cdot, \cdot, \ldots \rangle$ as well. Therefore,

$$tr(r)[p^+] = tr(r_1)[p^+]. \tag{3.25}$$

Since r_1 is a run of $\mathcal{N}_1$, Condition (3.13) of Lemma 3.1 is fulfilled for place p, i.e., $\langle m_0(p) \rangle \frown tr(r_1)[p^+] \succeq tr(r_1)[p^-]$. Because of Equalities (3.24) and (3.25), condition $\langle m_0(p) \rangle \frown tr(r)[p^+] \succeq tr(r)[p^-]$ is fulfilled for sequence r as well.

The treatment of output places $p \in P^o$ is analogous. There, roles of attributes p^+ and p^- are exchanged.

Case 3: $p \in \mathcal{N}_1.P^{io} \cap \mathcal{N}_2.P^{io}$:
Without restricting generality, we assume that place p is an output place of $\mathcal{N}_1$ and an input place of $\mathcal{N}_2$, i.e., $p \in \mathcal{N}_1.P^o \cap \mathcal{N}_2.P^i$. If we apply assumption '$o^+_{S_1} = i_{S_2}$' of Lemma 3.4 to place p, we have

$$tr(r_1)[p^+] = tr(r_2)[p^+]. \tag{3.26}$$

Since sequence r_2 is a run, and interface place p is initially empty, Condition (3.14) of Lemma 3.1 is valid for place p wrt. run r_2, i.e.,

$$\langle\cdot\rangle \frown tr(r_2)[p^+] \;\succeq\; tr(r_2)[p^-].$$

Together with statement (3.26), we infer

$$\langle\cdot\rangle \frown tr(r_1)[p^+] \;=\; \langle\cdot\rangle \frown tr(r_2)[p^+] \;\succeq\; tr(r_2)[p^-]. \tag{3.27}$$

Applying Lemma 3.3 (3.17) to attributes p^+, p^- implies

$$tr(r)[p^+] = tr(r_1)[p^+] \;\wedge\; tr(r)[p^-] = tr(r_2)[p^-].$$

Together with Statement (3.26), we obtain $\langle\cdot\rangle \frown tr(r)[p^+] \succeq tr(r)[p^-]$ wrt. sequence r which verifies Condition (3.14) of Lemma 3.1. □

The following lemma relates the internal i/o behavior of a net $\mathcal{N}$ to its externally observed i/o behavior. More precisely, as internal i/o behavior, we concern streams of $\mathcal{N}$ internally observed at its input/output places, i.e., streams $l^-_{S_{21}}$ and streams $l^+_{S_{12}}$ represented in Figure 3.19. Assumed there exists an observer $\mathcal{O}$ and a run r of $\mathcal{N} \circ \mathcal{O}$ with internal streams $l^-_{S_{21}}$, $l^+_{S_{12}}$. Then, the (external) i/o behavior relation $io(\mathcal{N})$ will contain all pairs (i_S, o_S) for which $i_S \succeq l^-_{S_{21}}$ and $o_S \preceq l^+_{S_{12}}$. Intuitively, an observer may produce more input than internally consumed, and may consume less output than internally produced. The implication of Lemma 3.5 is graphically illustrated in Figure 3.19.

Lemma 3.5 ***('Observable i/o behavior')*** *Let $\mathcal{N}$ be an interaction net, let $\mathcal{O}$ be an observer of $\mathcal{N}$, and r be a run of $\mathcal{N} \circ \mathcal{O}$. Then, for each stream tuples i_S, o_S, there exists an observer $\mathcal{O}_g$ of $\mathcal{N}$ and a run r' of $\mathcal{N} \circ \mathcal{O}_g$ with*

$$\begin{aligned} i_S \succeq tr'(r)[\tilde{l}^-_{S_{21}}] \;\wedge\; o_S \preceq tr'(r)[\tilde{l}^+_{S_{12}}] \;\Rightarrow\; \\ r'[S_{]\mathcal{N}[}] = r[S_{]\mathcal{N}[}] \;\wedge\; tr'(r')[\tilde{l}^+_{S_{21}}] = i_S \;\wedge\; tr'(r')[\tilde{l}^-_{S_{12}}] = o_S, \end{aligned} \tag{3.28}$$

where $\tilde{l}^+_{S_{12}} = (\mathscr{N}.P^o \cap \mathscr{O}.P^i)^+$, $\tilde{o}_S = \tilde{l}^-_{S_{12}} = (\mathscr{N}.P^o \cap \mathscr{O}.P^i)^-$, $\tilde{i}_S = \tilde{l}^+_{S_{21}} = (\mathscr{O}.P^o \cap \mathscr{N}.P^i)^+$, *and* $\tilde{l}^-_{S_{21}} = (\mathscr{O}.P^o \cap \mathscr{N}.P^i)^-$.

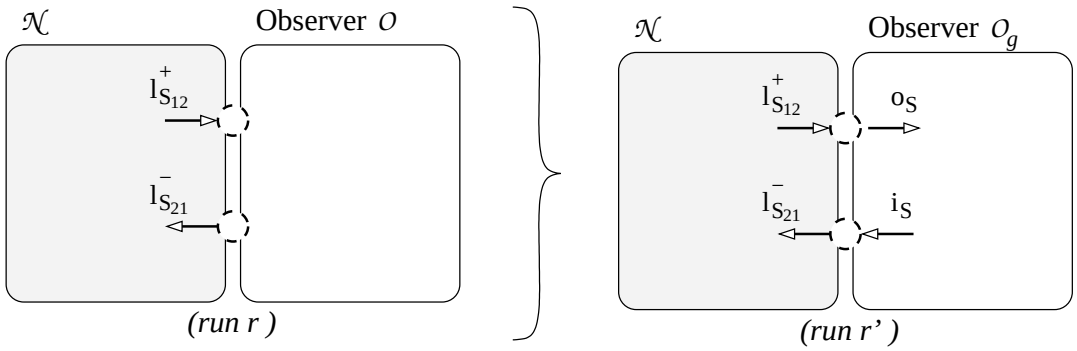


Figure 3.19: Graphical representation of Lemma 3.5: For each $i_S \succeq l^-_{S_{21}}$ and $o_S \preceq l^+_{S_{21}}$, there exists an observer $\mathscr{O}_g$ and an according run r'.

Proof We employ a general observer $\mathscr{O}_g$ of $\mathscr{N}$ specified by algorithm OBSERVER$_g$ (cf. page 47). By its construction, for each tuple $(i_S, o_S) : (\tilde{l}^+_{S_{21}}, \tilde{l}^-_{S_{12}})$, there exists a corresponding sequence of steps r_g of $\mathscr{O}_g$ with

$$tr(r_g)[\tilde{l}^+_{S_{21}}, \tilde{l}^-_{S_{12}}] \;=\; (i_S, o_S)\,.$$

We particularly consider tuple (i_S, o_S) with

$$i_S \succeq tr'(r)[\tilde{l}^-_{S_{21}}] \;\wedge\; o_S \preceq tr'(r)[\tilde{l}^+_{S_{12}}]\,. \tag{3.29}$$

We define $r' := r[S_{]\mathscr{N}[}] + r_g$. We then need to verify that (i) the consequence of Implication (3.28) is fulfilled, and (ii) r' is a run of $\mathscr{N} \circ \mathscr{O}_g$. By definition of r', $r'[S_{]\mathscr{N}[}] = r[S_{]\mathscr{N}[}]$. By Lemma 3.3 (3.17), we obtain

$$\begin{aligned} tr'(r')[\tilde{l}^+_{S_{21}}] &= tr'(r_g)[\tilde{l}^+_{S_{21}}] = i_S\,, \\ tr'(r')[\tilde{l}^-_{S_{12}}] &= tr'(r_g)[\tilde{l}^-_{S_{12}}] = o_S\,, \end{aligned}$$

since $\tilde{l}^+_{S_{21}}, \tilde{l}^-_{S_{12}} \in S_{]\mathscr{O}_g[}$. Together with Statement (3.29), it verifies Condition (3.28), i.e.,

$$tr'(r')[\tilde{l}^+_{S_{21}}] = i_S \;\wedge\; tr'(r')[\tilde{l}^-_{S_{12}}] = o_S\,. \tag{3.30}$$

It is left to show that r' is a run of $\mathscr{N} \circ \mathscr{O}_g$. By Lemma 3.3 (3.19), r' represents a sequence of steps of $\mathscr{N} \circ \mathscr{O}_g$. To verify that r' is a run, we will apply Lemma 3.1. Therefore, we have to confirm the validity of one of the conditions of Lemma 3.1 for each place $p \in (\mathscr{N} \circ \mathscr{O}_g).P$. We distinguish three cases: (1) non-i/o places of $\mathscr{N}$

and $\mathscr{O}_g$, (2) i/o places of $\mathscr{N} \circ \mathscr{O}_g$, and (3) i/o places of $\mathscr{N}$ and $\mathscr{O}_g$ which established connections. We omit cases (1) and (2), since they can be treated analogously to cases (1) and (2) at the proof of Lemma 3.4.

Case 3: $p \in \mathscr{N}.P^{io} \cap \mathscr{O}_g.P^{io}$:
Firstly, we consider place $p \in \mathscr{N}.P^{i} \cap \mathscr{O}_g.P^{o}$, i.e., p is an input place of $\mathscr{N}$ and an output place of $\mathscr{O}_g$. If we apply Statements (3.29) and (3.30) (left conjuncts) to place p, we obtain

$$tr'(r')[p^+] \succeq tr'(r)[p^-]\,, \tag{3.31}$$

since $p^+ \in \tilde{l}^+_{S_{21}}$ and $p^- \in \tilde{l}^-_{S_{21}}$. Because p is an interface place, and thus, initially empty, it verifies Condition (3.15) of Lemma 3.1.

Secondly, we consider place $p \in \mathscr{N}.P^{o} \cap \mathscr{O}_g.P^{i}$, i.e., p is an output place of $\mathscr{N}$ and an input place of $\mathscr{O}_g$. If we apply Statements (3.29) and (3.30) (right conjuncts) to place p, we obtain

$$tr'(r')[p^+] \succeq tr'(r)[p^-]\,, \tag{3.32}$$

since $p^+ \in \tilde{l}^+_{S_{12}}$ and $p^- \in \tilde{l}^-_{S_{12}}$. Because p is an interface place, and thus, initially empty, it verifies Condition (3.15) of Lemma 3.1. □

We are now ready to proof Theorem 3.1:

Proof (of Theorem 3.1) Equality (3.10) can be rewritten as an equivalence:

$$\forall x_S.\ \ x_S \in io(\mathscr{N}_1 \circ \mathscr{N}_2) \ \ \Leftrightarrow \ \ x_S \in io(\mathscr{N}_1) \bowtie io(\mathscr{N}_2)\,.$$

If we replace function $io()$ and operator $\bowtie$ by their definitions, this equivalence corresponds to:

$$\begin{aligned}
&\forall i_{S_1}, i_{S_2}, o_{S_1}, o_{S_2}.\\
&\quad \exists \mathscr{O}.\ (i_{S_1}, i_{S_2}, o_{S_1}, o_{S_2}) \in tr'(\mathscr{N}_1 \circ \mathscr{N}_2 \circ \mathscr{O})[\cdot] \ \Leftrightarrow\\
&\qquad \exists \mathscr{O}_1, \mathscr{O}_2, l^+_{S_{12}}, l^-_{S_{12}}, l^+_{S_{21}}, l^-_{S_{21}}.\ l^+_{S_{12}} = l^-_{S_{12}} \ \wedge\ l^+_{S_{21}} = l^-_{S_{21}} \ \wedge\\
&\qquad\qquad (i_{S_1}, l^+_{S_{21}}, o_{S_1}, l^-_{S_{12}}) \in tr'(\mathscr{N}_1 \circ \mathscr{O}_1)[\cdot] \ \wedge\\
&\qquad\qquad (i_{S_2}, l^+_{S_{12}}, o_{S_2}, l^-_{S_{21}}) \in tr'(\mathscr{N}_2 \circ \mathscr{O}_2)[\cdot] \ ,
\end{aligned} \tag{3.33}$$

if '$(x_S) \in tr(\mathscr{N})[\cdot]$' denotes that $x_S : S$ is an element of the accordingly projected trace $tr'(\mathscr{N})[S]$. We denote the left-hand side of equivalence (3.33) by (L) and the right-hand side by (R).

Part (i): (L) $\Rightarrow$ (R)

(L) implies that there exists a run r that generates trace $(i_{S_1}, i_{S_2}, o_{S_1}, o_{S_2}) = tr'(r)[\cdot]$. If we extend the projection, we have

$$(i_{S_1}, i_{S_2}, o_{S_1}, o_{S_2}, l'^{+}_{S_{12}}, l'^{-}_{S_{12}}, l'^{+}_{S_{21}}, l'^{-}_{S_{21}}) = tr'(r)[\cdot], \tag{3.34}$$

where the l'_S denote local streams observed at run r according to Figure 3.17. Note that there is a logical difference between local streams l_S in Statement (3.33) and local streams l'_S although their data types correspond to one another. While l_S denotes streams observed at runs of $\mathscr{N}_1 \circ \mathscr{O}_1$ and $\mathscr{N}_2 \circ \mathscr{O}_2$ respectively, l'_S denotes streams observed at run r of $\mathscr{N}_1 \circ \mathscr{N}_2 \circ \mathscr{O}$.

By applying Lemma 3.1 (3.15) to local streams l'_S observed at run r, we obtain

$$\begin{array}{l} l'^{-}_{S_{12}} \preceq l'^{+}_{S_{12}} \ \wedge \\ l'^{-}_{S_{21}} \preceq l'^{+}_{S_{21}}. \end{array} \tag{3.35}$$

Since $\mathscr{N}_1 \circ (\mathscr{N}_2 \circ \mathscr{O})$ exists, $\mathscr{O}_1 := (\mathscr{N}_2 \circ \mathscr{O})$ is an observer of $\mathscr{N}_1$. Therefore, r is a run of $\mathscr{N}_1 \circ \mathscr{O}_1$ as well. Lemma 3.5 ensures the existence of an observer $\mathscr{O}'_1$, a run r_1 of $\mathscr{N}_1 \circ \mathscr{O}'_1$, and stream tuples $l''^{+}_{S_{21}}$, $l''^{-}_{S_{12}}$ with

$$\begin{array}{l} l''^{+}_{S_{21}} = l'^{-}_{S_{21}} \ \wedge \ l''^{-}_{S_{12}} = l'^{-}_{S_{12}} \ \Rightarrow \\ (i_{S_1}, l''^{+}_{S_{21}}, o_{S_1}, l''^{-}_{S_{12}}, l'^{+}_{S_{12}}, l'^{-}_{S_{21}}) = tr(r_1)[\cdot], \end{array} \tag{3.36}$$

since $l'^{-}_{S_{12}} \preceq l'^{+}_{S_{12}}$ (cf. Statement (3.35)). Analogously, we obtain an observer $\mathscr{O}'_2$, a run r_2 of $\mathscr{N}_2 \circ \mathscr{O}'_2$, and stream tuples $l''^{+}_{S_{12}}$, $l''^{-}_{S_{21}}$ with

$$\begin{array}{l} l''^{+}_{S_{12}} = l'^{-}_{S_{12}} \ \wedge \ l''^{-}_{S_{21}} = l'^{-}_{S_{21}} \ \Rightarrow \\ (i_{S_2}, l''^{+}_{S_{12}}, o_{S_2}, l''^{-}_{S_{21}}, l'^{-}_{S_{12}}, l'^{+}_{S_{21}}) = tr(r_2)[\cdot], \end{array} \tag{3.37}$$

since $l'^{-}_{S_{21}} \preceq l'^{+}_{S_{21}}$ (cf. Statement (3.35)).

Together, Statements (3.36) and (3.37) provide the consequence

$$\begin{array}{rl} \exists \mathscr{O}'_1, \mathscr{O}'_2, l''^{+}_{S_{12}}, l''^{-}_{S_{12}}, l''^{+}_{S_{21}}, l''^{-}_{S_{21}}. \ l''^{+}_{S_{12}} = l''^{-}_{S_{12}} \ \wedge \ l''^{+}_{S_{21}} = l''^{-}_{S_{21}} & \wedge \\ (i_{S_1}, l''^{+}_{S_{21}}, o_{S_1}, l''^{-}_{S_{12}}) \in tr'(\mathscr{N}_1 \circ \mathscr{O}'_1)[\cdot] & \wedge \\ (i_{S_2}, l''^{+}_{S_{12}}, o_{S_2}, l''^{-}_{S_{21}}) \in tr'(\mathscr{N}_2 \circ \mathscr{O}'_2)[\cdot] & , \end{array}$$

which corresponds to (R).

Part (ii): (L) $\Leftarrow$ (R)

(R) implies that there exist runs r_1 and r_2 of $\mathscr{N}_1 \circ \mathscr{O}_1$ and $\mathscr{N}_2 \circ \mathscr{O}_2$ respectively that generate traces

$$
\begin{array}{l}
(i_{S_1}, l^+_{S_{21}}, o_{S_1}, l^-_{S_{12}}, l^+_{S_{12}}, l^-_{S_{21}}) = tr'(r_1)[\cdot] \ \wedge \\
(i_{S_2}, \underline{l}^+_{S_{12}}, o_{S_2}, \underline{l}^-_{S_{21}}, \underline{l}^+_{S_{21}}, \underline{l}^-_{S_{12}}) = tr'(r_2)[\cdot] \ \wedge \\
\underline{l}^+_{S_{12}} = l^-_{S_{12}} \ \wedge \ l^+_{S_{21}} = \underline{l}^-_{S_{21}},
\end{array}
\tag{3.38}
$$

where we distinguish local streams observed at run r_2 by underscore. Concerning run r_1 of $\mathscr{N}_1 \circ \mathscr{O}_1$, Lemma 3.5 ensures the existence of an observer $\mathscr{O}'_1$, a run r'_1 of $\mathscr{N}_1 \circ \mathscr{O}'_1$ with $r'_1[S_{]\mathscr{N}_1[}] = r_1[S_{]\mathscr{N}_1[}]$, and a stream tuple $l'^+_{S_{21}}$ with

$$
\begin{array}{l}
l'^+_{S_{21}} = \underline{l}^+_{S_{21}} \ \Rightarrow \\
(i_{S_1}, l'^+_{S_{21}}, o_{S_1}, l^-_{S_{12}}, l^+_{S_{12}}, l^-_{S_{21}}) = tr(r'_1)[\cdot].
\end{array}
\tag{3.39}
$$

Note that assumption $l'^+_{S_{21}} \succeq l^-_{S_{21}}$ of Lemma 3.5 is satisfied, since

$$
\begin{array}{ll}
l'^+_{S_{21}} = \underline{l}^+_{S_{21}} & \text{(by assignment in (3.39)),} \\
\underline{l}^+_{S_{21}} \succeq \underline{l}^-_{S_{21}} & \text{(by Lemma 3.1 (3.15) applied to run } r_2\text{),} \\
\underline{l}^-_{S_{21}} = l^+_{S_{21}} & \text{(by Statement (3.38)), and} \\
l^+_{S_{21}} \succeq l^-_{S_{21}} & \text{(by Lemma 3.1 (3.15) applied to run } r_1\text{).}
\end{array}
$$

Analogously, we obtain an observer $\mathscr{O}'_2$, a run r'_2 of $\mathscr{N}_2 \circ \mathscr{O}'_2$ with $r'_2[S_{]\mathscr{N}_2[}] = r_2[S_{]\mathscr{N}_2[}]$, and a stream tuple $\underline{l}'^+_{S_{12}}$ with

$$
\begin{array}{l}
\underline{l}'^+_{S_{12}} = l^+_{S_{12}} \ \Rightarrow \\
(i_{S_2}, \underline{l}'^+_{S_{12}}, o_{S_2}, \underline{l}^-_{S_{21}}, \underline{l}^+_{S_{21}}, \underline{l}^-_{S_{12}}) = tr(r'_2)[\cdot].
\end{array}
\tag{3.40}
$$

We may assume that observers $\mathscr{O}'_1$ and $\mathscr{O}'_2$ are general observers specified by algorithm OBSERVER$_g$ (cf. page 47). To permit composition of runs r'_1, r'_2 by application of Lemma 3.4, we employ the following perspective. As illustrated in Figure 3.20, we decompose both observers $\mathscr{O}'_1 = \mathscr{O}'^e_1 \circ \mathscr{O}'^i_1$, and $\mathscr{O}'_2 = \mathscr{O}'^e_2 \circ \mathscr{O}'^i_2$ into an external part $\mathscr{O}^e$ and an internal part $\mathscr{O}^i$. Decomposition of $\mathscr{O}'_1$ and $\mathscr{O}'_2$ is based on i/o places:

$$
\begin{array}{rcl}
\mathscr{O}'^i_1.P \ = \ \mathscr{O}'^i_2.P & := & \mathscr{N}_1.P^{io} \cap \mathscr{N}_2.P^{io}, \\
\mathscr{O}'^e_1.P & := & \mathscr{O}'_1.P \setminus \mathscr{O}'^i_1.P, \\
\mathscr{O}'^e_2.P & := & \mathscr{O}'_2.P \setminus \mathscr{O}'^i_2.P.
\end{array}
$$

Attached arcs, transition, etc. belong to the sub-net that contains corresponding places. Thus, $\mathscr{O}'^i_1$, $\mathscr{O}'^i_2$ address connected places of $\mathscr{N}_1 \circ \mathscr{N}_2$ and $\mathscr{O}'^e_1$, $\mathscr{O}'^e_2$ address remaining unconnected places. It is straightforward to verify that $\mathscr{O}'_1 = \mathscr{O}'^e_1 \circ \mathscr{O}'^i_1$ and $\mathscr{O}'_2 = \mathscr{O}'^e_2 \circ \mathscr{O}'^i_2$, since the sub-nets are disjunctive interaction nets. We apply Lemma 3.4 through the following assignment (cf. Figures 3.20 and 3.18):

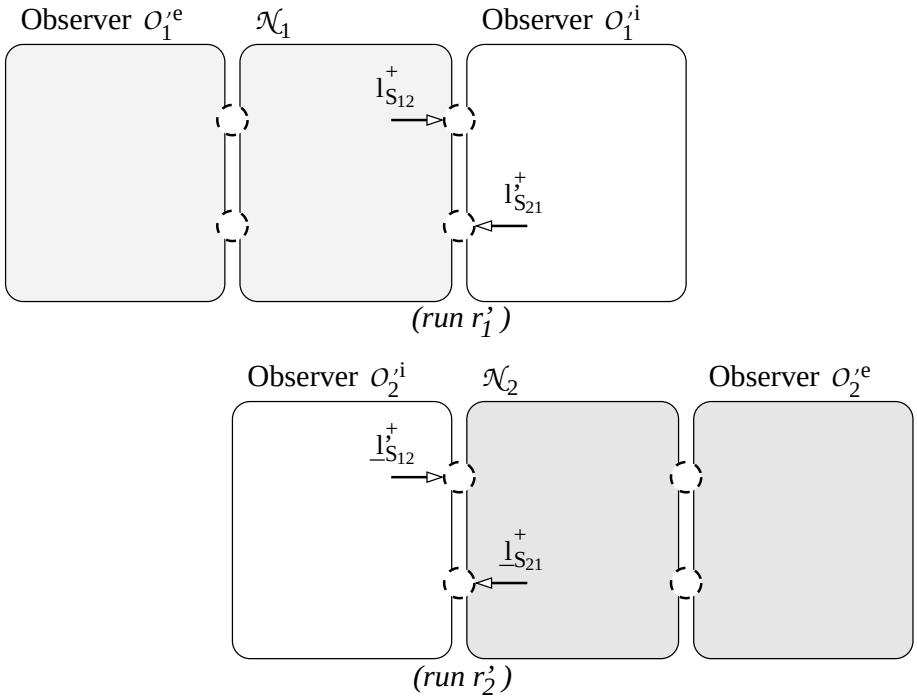


Figure 3.20: An adapted perspective onto $\mathcal{N}_1 \circ \mathcal{O}_1'$ and $\mathcal{N}_2 \circ \mathcal{O}_2'$ by decomposition of observers

$$\begin{aligned}
\mathcal{N}_1 &\hookleftarrow \mathcal{N}_1 \circ \mathcal{O}_1^e, \\
\mathcal{N}_2 &\hookleftarrow \mathcal{N}_2 \circ \mathcal{O}_2^e, \\
\mathcal{O}_1 &\hookleftarrow \mathcal{O}_1^i, \\
\mathcal{O}_2 &\hookleftarrow \mathcal{O}_2^i, \\
r_1 &\hookleftarrow r_1', \\
r_2 &\hookleftarrow r_2'.
\end{aligned}$$

Assumption $l'^+_{S_{21}} = \underline{l}^+_{S_{21}} \wedge \underline{l}'^+_{S_{12}} = l^+_{S_{12}}$ of Lemma 3.4 is provided by Statements (3.39) and (3.40). (To be precise, the shifted traces, must be replaced by their according plain traces in Lemma 3.4. However, both equalities are satisfied by the according plain traces as well.) Lemma 3.4 implies that $r' := r_1'[S_{]\mathcal{N}_1[}] + r_2'[S_{]\mathcal{N}_2[}]$ is a run of $(\mathcal{N}_1 \circ \mathcal{O}_1'^{e}) \circ (\mathcal{N}_2 \circ \mathcal{O}_2'^{e})$. By definition of r_1' and r_2', we obtain

$$(i_{S_1}, i_{S_2}, o_{S_1}, o_{S_2}, l^+_{S_{12}}, \underline{l}^-_{S_{12}}, \underline{l}^+_{S_{21}}, l^-_{S_{21}}) = tr'(r')[\cdot],$$

and thus, we obtain (L):

$$\exists \mathcal{O}.\ (i_{S_1}, i_{S_2}, o_{S_1}, o_{S_2}) \in tr'(\mathcal{N}_1 \circ \mathcal{N}_2 \circ \mathcal{O})[\cdot],$$

with observer $\mathcal{O} := \mathcal{O}_1'^{e} \circ \mathcal{O}_2'^{e}$. □

3.3 Components in Environment

Commonly, components are not autonomous sub-systems, but open system which interact with their environment. Thereby, we consider an environment $\mathscr{E}$ of a component $\mathscr{C}$ as the collection of all components connected to $\mathscr{C}$. Syntactic questions as: "Does the syntactic interface of a component $\mathscr{C}$ match its environment?" are determined by the composition operator $\otimes$. Thereby, a positive answer to the question corresponds to the existence of composition $\mathscr{C} \otimes \mathscr{E}$. However, to reason about "meaningfulness" of a composition, we additionally have to consider the dynamics of composition.

Generally, we may view composition from two perspectives: (i) from the perspective of a particular component, and (ii) from the perspective of an environment where a component is introduced into (or replaced). Both perspectives arise distinguished questions:

Component perspective:

- **(Q1)** Which assertions must be imposed onto an environment, such that a component functions as intended?
- **(Q2)** Which properties are provided by a component within an appropriate environment?

Environmental perspective:

- **(Q3)** How can we verify whether an environment satisfies assertions imposed by a component (to be embedded)?
- **(Q4)** Under which conditions may we replace components by different versions within an environment?

As will be shown later, answers to these questions are closely related to one another. In addition, these answers closely relate to our approach of providing formal solutions to interaction patterns in terms of interaction nets. As interaction nets can be considered as components (cf. Section 3.2), the above questions are applicable to this context. In particular, question (Q3) indicates if different versions of pattern realizations may replace one another within complex compositions.

To clarify the above questions, we consider an example component specification which provides a service request queue (illustrated in Figure 3.21). The intention of the request queue is to provide access of a single service provider to several clients, in particular, to delegate client requests to the provider and to inform clients about responses. There are several requirements which an according queue component may assume to function properly. For example, the service provider is assumed (i) to respond to initiated requests within a finite time period, and (ii) to provide an according request identifier of the performed request. At a restricted setting, the queue component might

furthermore require that an individual client might send a subsequent request only after a response is received. If according requirements are fulfilled, the queue component guarantees properties as, for example, each client request will be delegated to the provider and answered within a finite time period.

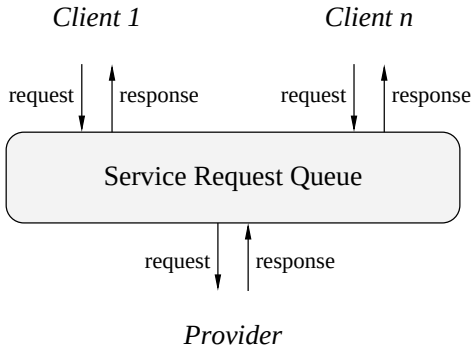


Figure 3.21: Illustration of a service request queue

We will refer to properties a component requires from its environment as *assertions.* Thereby, assertions represent dynamic constraints imposed onto the environment. Note that properties and assertions are commonly associated with a particular (sub-)interface only. At the example above, different assertions are imposed wrt. the (sub-)interface to the service provider and wrt. (sub-)interfaces to the clients.

As will be shown in Section 3.3.2, the questions posed above are closely related to the notion of component refinement. In particular, the replacement of components by refined versions provides desired properties. According to formalization of interaction patterns, it motivates an opportunity to derive specialization hierarchies on interaction nets. In contrast to pattern hierarchies derived from the intuition of a designer, the formal notion of refinement permits an inference of specialization relationship based on the actual behavior.

3.3.1 Component Refinement

Broy et. al. [BS01] propose three refinement relations on components: (i) behavioral refinement, (ii) interface refinement, and (iii) conditional refinement. Thereby, conditional refinement subsumes the others, and interface refinement subsumes behavioral refinement. The most convenient (but least expressive) refinement relation is the behavioral refinement. It provides the feature that if we replace a sub-component of a complex composition by a refined version, then the resulting complex component is a refinement of the original as well. This feature is not guaranteed for the other refinement relations. However, there exist certain sub-classes of interface refinement which enable similar statements. We will briefly introduce behavioral refinement and

interface refinement in the following. Regarding conditional refinement, we refer to [BS01].

Behavioral Refinement

Roughly, behavioral refinement can be stated as follows. A component specification S_2 is a behavioral refinement of a component specification S_1, if both possess the same syntactic interface, and each pair of input and output stream that occurs in the behavior of S_2 also occurs in the behavior of S_1. At the logical level, this refinement relation can be expressed by logical implication.

Definition 3.13 *Let S_1 and S_2 be component specifications with the same syntactic interface. Relation $\rightsquigarrow$* of behavioral refinement *is defined by the equivalence*

$$(S_1 \rightsquigarrow S_2) \quad \Leftrightarrow \quad (\llbracket S_2 \rrbracket \Rightarrow \llbracket S_1 \rrbracket)\,.$$

Thereby, S_2 refers to the refined specification. According to system development, behavioral refinement supports reduction of underspecification. For example, a service request queue Q_2 which prioritizes client requests in a specific way represents a behavioral refinement of a queue Q_1 which does not specify an order of processing requests. Generally, behavioral refinement can be characterized by the subset relationship on according i/o behavior relations, i.e.,

$$(S_1 \rightsquigarrow S_2) \quad \Rightarrow \quad io(S_2) \subseteq io(S_1)\,.$$

An essential feature of component refinement is that component composition $\otimes$ is monotonic wrt. behavioral refinement (cf. [BS01]). More precisely,

$$S_1 \rightsquigarrow S_1' \wedge S_2 \rightsquigarrow S_2' \quad \Rightarrow \quad S_1 \otimes S_2 \rightsquigarrow S_1' \otimes S_2'\,.$$

It implies that if we replace sub-components of a system by refined versions, then the new system represents a refinement of the original.

Interface Refinement

Interface refinement is a generalization of behavioral refinement. As the name suggests, besides the behavior specification, the interface may be adapted. It is defined as follows:

Definition 3.14 *Let S_1, S_2, D, and U be component specifications with the following syntactic interfaces*

$$S_1 \in (I_1 \rhd O_1), \quad S_2 \in (I_2 \rhd O_2), \quad D \in (I_1 \rhd I_2), \quad U \in (O_2 \rhd O_1)\,.$$

Relation $\overset{(D,U)}{\rightsquigarrow}$ of interface refinement *is defined as follows:*

$$S_1 \overset{(D,U)}{\rightsquigarrow} S_2 \;\Leftrightarrow\; S_1 \rightsquigarrow (D \otimes S_2 \otimes U) \;\Leftrightarrow\; [\![D \otimes S_2 \otimes U]\!] \Rightarrow [\![S_1]\!]\,.$$

Thereby, S_2 refers to the refined specification. Component specifications D and U are referred to as *representation specifications*. As illustrated in Figure 3.22, D and U provide a translation between i/o streams of the abstract component S_1 and the refined component S_2. Intuitively, *downwards relation D* converts input streams of S_1 such that S_2 can process them, and *upwards relation U* reconverts output stream of S_2 such that they correspond to output streams of S_1. Thus, interface refinement can be understood as behavioral refinement modulo an interface conversion.

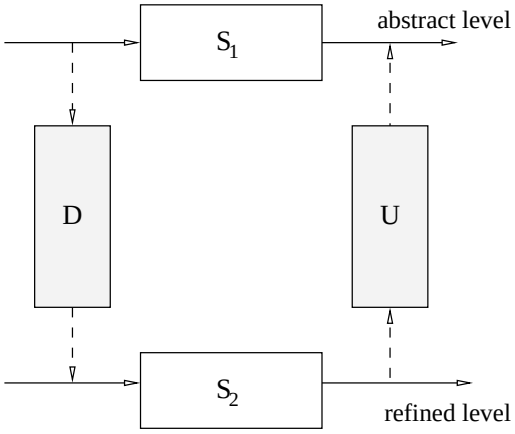


Figure 3.22: Illustration of interface refinement

According to system development, interface refinement corresponds to such activities as:

- changing data types of messages, for example, representing messages by more adequate types,
- changing the granularity of interaction, for example, replacing an interaction realized by a single step by a multiple step interaction,
- modifying the communication structure, for example, representing one channel by several, or vice versa,

- adaptation, for example, changing the interface so that it satisfies environmental changes,
- adding services, for example, by introducing additional input and output messages,
- inclusion of exception handling.

A common example of interface refinement is the conversion of message types. For example, a component which realizes term rewriting consume and produces terms. Thereby, message types of input and output channels may be character strings. However, a deployment of this component into an environment might require a representation of terms by means of XML trees. An according component therefore results from an interface refinement of the original string processing term rewriting component. Thereby, the corresponding downwards relation D converts terms represented by character strings into their tree representations, and upwards relation U computes the inverse.

In contrast to behavioral refinement, the composition operator $\otimes$ is not monotonic wrt. interface refinement in the general case. However, by imposing certain assumptions, component composition $\otimes$ is monotonic wrt. interface refinement as well. An assumption of practical relevance requires restrictions on representation specifications D and U. It assumes that D and U are so-called *refinement pairs*, i.e., D must represent the inverse function of U and vice versa. For example, representation specifications D and U which convert terms between string and tree representation represent a refinement pair.

A general opportunity to replace components by according refined versions resulting from interface refinements is to provide representation specifications D and U. The introduction of the refined component then imposes the additional design task of adapting its environment according to specifications D and U.

3.3.2 Component Properties and Assertions

The specification of behavioral properties of components can be based on logical formulas. Thereby, an extension $\mathscr{R}_p$ of a formula p (i.e., all valid assignments of p) reflects according behavioral restrictions. If a component permits an input/output pair outside extension $\mathscr{R}_p$, this component does not satisfy property p. Furthermore, properties can be used to define assertions. Thereby, an assertion is specified by a property p_a. It is interpreted as: if a component $\mathscr{C}$ requires an assertion p_a, then a connected component $\mathscr{C}'$ has to satisfy property p_a. In other words, assertions represent properties imposed onto connected components.

To associate properties with interfaces, we specify a property by a particular component specification $\mathscr{C}_p$. Besides body formula p, component specification $\mathscr{C}_p$ associates

p with a syntactic interface $\mathscr{I}_p = (I_S \triangleright O_S)$ including respective type constraints. For readability, we will apply representations (i) p, (ii) p wrt. interface $\mathscr{I}_p$, as well as (iii) $\mathscr{C}_p$ to refer to a property.

Thereby, the questions posed at the beginning of Section 3.3 can be answered as follows (for readability, we slightly permuted them to Q1, Q3, Q2, Q4):

(Q1) Which assertions must be imposed onto an environment, such that a component functions as intended?

The specification of an assertion corresponds to a component specification $\mathscr{C}_a$ which states according requirements. As an example, we reconsider the service request queue illustrated at the beginning of Section 3.3. It requires that a connected service provider responses to requests within a finite time period. In addition, we might require that the order of responses corresponds to the order of requests (if multiple requests are processed by a provider at the same time). This assertion can be defined by a component specification $\mathscr{C}_a$ with an interface that provides a single input and a single output channel (cf. Figure 3.23).

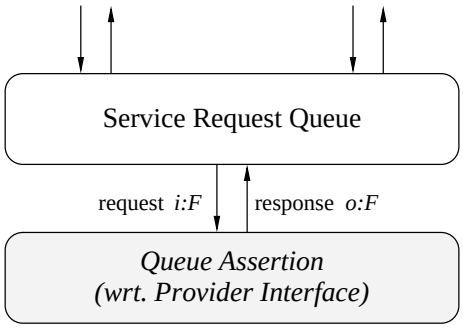


Figure 3.23: Illustration of an assertion required by a request queue

It can be formally represented by the following component specification:

Queue Assertion (wrt. Provider Interface) [timed]
in $i : F$ out $o : F$
$\forall k.\ \overline{i}.k[1] = \overline{o}.k[1] \ \vee \ k > \#\overline{i}$

Thereby, we define type $F := tuple(int, string)$ denoting pairs of integers and character strings. We notate an access to tuple components by postfix '$[j]$' where j indicates

the position within a tuple. According to message type F, we use integers to represent unique request identifiers and strings to represent requests as well as responses. Thereby, above specification ensures that each request is eventually responded, and the order of requests corresponds to the order of associated responses. Note that assertion $\mathscr{C}_a$ does not specify the actual service function of a particular provider. It corresponds to the intention of the service request queue to serve any kind of service provider (assumed above assertion will be satisfied).

(Q3) How can we verify whether an environment satisfies assertions imposed by a component (to be embedded)?

The verification whether an environment satisfies required assertions is reduced to the test of property satisfaction. More precisely, connected components $\mathscr{C}$ must satisfy required properties. The decision problem whether a component $\mathscr{C}$ satisfies a property p is characterized:

(i) at the logical level: by logical implication, and

(ii) at the component level: by component refinement, more precisely, a component $\mathscr{C}$ satisfies property $\mathscr{C}_p$, iff $\mathscr{C}_p \rightsquigarrow \mathscr{C}$, i.e., iff $\mathscr{C}$ represents a behavioral refinement of $\mathscr{C}_p$,

assumed that syntactic interfaces of $\mathscr{C}$ and $\mathscr{C}_p$ are identical. At a general perspective, we must permit the case, where interface $\mathscr{I}_p$ of property specification $\mathscr{C}_p$ corresponds to a subset of interface $\mathscr{I}$ of $\mathscr{C}$ only. At the queue example, this case occurs, if a service provider possesses further sub-interfaces, for example, to request lower-level services in turn. The interface $\mathscr{I}_p$ of the queue assertion then is a proper subset of interface $\mathscr{I}$ of the service provider. As a generalization of this situation, we additionally assume that component $\mathscr{C}$ itself imposes further assertions a onto its environment. It represents a generalization, since "no assertion" can always be represented by the trivial assertion, i.e., an empty body formula together with an according interface. Thereby, an empty body formula corresponds to '*true*', and therewith, this component exhibits an unrestricted behavior at the specified interface. As illustrated in Figure 3.24, the behavior of component $\mathscr{C}$ wrt. interface $\mathscr{I}_p$ is then determined by the composition $\mathscr{C} \otimes \mathscr{C}_a$.

Therefore, the property satisfaction test extends to

(ii') A component $\mathscr{C}$ (which requires assertion a) satisfies a property p, iff $\mathscr{C}_p \rightsquigarrow (\mathscr{C} \otimes \mathscr{C}_a)$, i.e., iff $\mathscr{C} \otimes \mathscr{C}_a$ is a behavioral refinement of $\mathscr{C}_p$.

Sometimes behavioral refinement cannot be achieved. However, the weaker notion of interface refinement is often applicable. As discussed in Section 3.3.1, a corresponding solution is to determine representation specifications D and U and to accordingly adapt the environment.

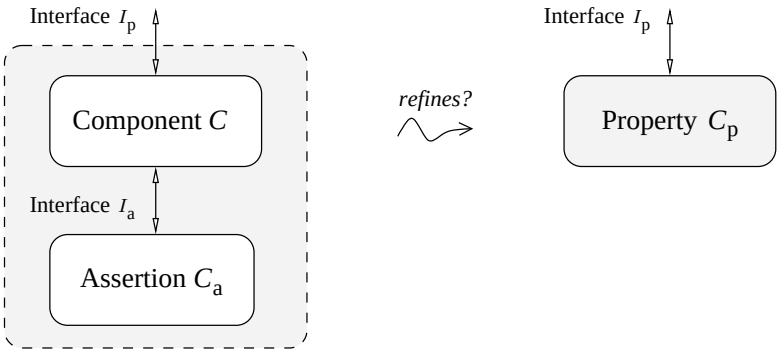


Figure 3.24: Property satisfaction test at the existence of assertions

(Q2) Which properties are provided by a component within an appropriate environment?

Assumed $\mathscr{C}$ represents a component and $\mathscr{C}_a$ its required assertion. According to the answer to question (Q3), property satisfaction corresponds to behavioral refinement wrt. composition $\mathscr{C} \otimes \mathscr{C}_a$. Therefore, an according test must be performed for each property of interest.

(Q4) Under which conditions may we replace components by different versions within an environment?

This question is answered by Broy et. al. [BS01] (cf. Section 3.3.1). Component versions which result from behavioral refinements may be replaced without changing the environment. The complete system behavior then exhibits a behavioral refinement as well. In the case of interface refinement, representation functions D and U must be provided to adapt the environment accordingly.

Because of Theorem 3.1, answers to above questions analogously apply to interaction net specifications. Reconsider, for example, question (Q2). In terms of interaction nets, it states: "Which properties are provided by an interaction net $\mathscr{N}$ within an appropriate environment?" Firstly, an appropriate environment must be specified. It is generally realized by a component $\mathscr{C}_a$ which reflects required assertions of interaction net $\mathscr{N}$. Thereby, component $\mathscr{C}_a$ can be specified by an interaction net $\mathscr{N}_a$ which reflects assertion a. Afterwards, properties of interaction net $\mathscr{N}$ "within an appropriate environment" are derived from the composition $\mathscr{N} \circ \mathscr{N}_a$. More precisely, as behavior of interaction nets is based on the notion of observer, properties are derived from the composition of $\mathscr{N} \circ \mathscr{N}_a$ with a (general) observer.

3.4 Concluding Remarks

Although properties in terms of logical formulas provide a rather compact behavior representation, it is often desired to directly observe and analyze the external behavior in terms of actual messages. For example, it supports testing of finite scenarios as well as improves the understanding of the behavior, particularly, if external components have to be deployed. While the behavior definition (cf. Definition 3.8) yields a desired black-box semantics, it is not always appropriate for human analysis. The reason is that input/output streams of black-box net behavior are interfered by sub-sequences of empty messages which decreases readability. It is caused by interaction nets which include complex internal computations. Thereby, the generation of corresponding output messages is basically delayed. Therefore, a transformation function on the i/o behavior relation would be beneficial which provides a more compact view. The adapted representation should abstract from the actual time an interaction net requires for the computations. A straightforward solution is achieved by removing "slices" of empty messages from input/output streams. For example, consider the following element of the i/o behavior relation of an interaction net.

$$\begin{array}{rl} i: & \langle 12 \;\; \cdot \;\; \cdot \;\; 17 \;\; 5 \;\; \cdot \;\; \cdot \;\; \cdot \;\; 3 \;\; \ldots \rangle \\ o: & \langle \cdot \;\; \cdot \;\; \cdot \;\; \cdot \;\; \cdot \;\; 12 \;\; \cdot \;\; 17 \;\; 5 \;\; \ldots \rangle \end{array}$$

Its compact representation removes empty messages at positions 2, 3, and 7 completely:

$$\begin{array}{rl} i: & \langle 12 \;\; 17 \;\; 5 \;\; \cdot \;\; \cdot \;\; 3 \;\; \ldots \rangle \\ o: & \langle \cdot \;\; \cdot \;\; \cdot \;\; 12 \;\; 17 \;\; 5 \;\; \ldots \rangle \end{array}$$

Note that the causality between input and output messages is maintained by the transformation. Thereby, it can be interpreted as a transformation which abstracts from "net time", whereby net time is considered as advancing at each occurrence of a step. However, at a more general perspective, the composite behavior should be derivable from the compact representations as well. More precisely, a transformation function f respecting composition needs to satisfy:

$$f(io(\mathcal{N}_1) \circ io(\mathcal{N}_2)) \;\; = \;\; f(io(\mathcal{N}_1)) \bowtie_f f(io(\mathcal{N}_2))$$

with a possibly adapted behavior composition $\bowtie_f$. Note that the introduced transformation does not provide this property.

Chapter 4

Application to Information Services

In this chapter, we propose an approach how the component framework introduced in Chapters 2 and 3 can be applied to user interaction. Thereby, we focus on the domain of information services. For this purpose, we introduce an abstraction called 'ui-components' and an according architecture based on interaction nets. For readability, we present the details of the component architecture at a top-down approach. In Section 4.1, we illustrate the big picture of the approach. Sections 4.2 and 4.3 provide a more detailed view onto the introduced notions of *ui-components* and *ui-composition components*. Finally, Section 4.4 demonstrates how the architecture can be realized on the basis of interaction nets. In addition, in Section 4.5, we discuss how far aspects of user interface quality can be verified, and in Section 4.6, we provide an according case study which abstractly specifies interactive catalogs.

4.1 A Component-Based Architecture

We apply the perspective that interaction specification of information services is based on the composition of elementary specifications. According to elementary interaction specification, we employ interaction nets as introduced in Section 2.2. They basically specify system responses to user input together with according context transitions. Therefore, we call them *ui-components* — where "ui" is derived from "user interaction". As illustrated in Figure 4.1, a ui-component essentially possesses two interfaces "ui-spec" and "context". Note that arrows used in the figure represent the main data flow only. In addition, they do not necessarily denote single channels, but may correspond to multiple, semantically related channels. According to interfaces which reflect a client/server policy, we distinguish the direction of service requests by filled arrows. The opposite direction of service provision is represented as usual by hollow arrows.

Interface "ui-spec": Through the interface "ui-spec", a ui-component provides a (declarative) specification of the next *dialog step*. In general, dialog steps are

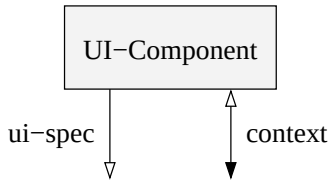


Figure 4.1: UI-components provide two interfaces

considered as elementary user activities as, for example, initiating a next event, filling a contract, or initiating a print job. At declarative approaches, these (elementary) dialog steps are specified by the application. A common example represents HTML code. It declaratively specifies the system output together with activated user events. Subsequently, the interaction specification is interpreted by a user interface controller, in the case of HTML, a Web browser. As a generalization, interface ”ui-spec”, is not restricted to a particular language as, for example, HTML, but may provide interaction specifications which serve arbitrary user interfaces. The specification provided through ”ui-spec” generally comprises system output together with activated user events — possibly enriched by meta data which primarily reflect style information. Note that we consider two levels of interaction specification (i) at design time and (ii) at runtime. An interaction specification at design time specifies user interaction of the complete system. In our approach, it is defined by interaction nets or components in general. An interaction specification at runtime specifies user interaction of a single dialog step which is afterwards interpreted by a user interface controller. For readability, we will apply the notion of a dialog step to indicate an according user activity as well as to indicate its corresponding declarative interaction specification as far as confusion is excluded.

Interface ”context”: Through interface ”context”, a ui-component may publish its internal context and may request external context. This interface provides the basis for the resolution of inter-component context dependencies. Besides context in general, initiated user events are conveyed through this interface as well. More precisely, user events are associated with specific contexts. Thereby, interface ”context” serves two major purposes: initiation of actions in terms of context publication, and receiving of user events. Without pre-drawing the details, we consider the context of an ui-component as a collection of named values of any data type. We sometimes also refer to a single element of the collection as context.

The overall behavior of ui-components then corresponds to (i) receiving (user) events, (ii) adapting the internal context (together with a possible context publication), and (iii) providing according output to the user in terms of a dialog (sub-)specification. Accord-

ing to the problem of identifying appropriate ui-components, interaction patterns may provide a promising basis. Since interaction patterns may correspond to different realizations, we commonly have to associate a single pattern with several versions of ui-components. Depending on application requirements, an appropriate version can be selected and, if required, be refined.

To derive complex ui-components from elementary ones, we propose so-called *ui-composition components* (cf. Figure 4.2) — as previously motivated at the end of Section 2.4. They basically realize two responsibilities:

Composition of dialog steps: Each elementary ui-component provides an interaction specification of the next dialog step. There exist different opportunities to integrate these specifications. Besides a straightforward concatenation of specifications, sophisticated integration rules might be applied. It can be compared to XML transformation languages as, for example, XSLT which given specifications (in terms of XML documents) together with a set of style rules, they produce an integrated specification.

Specification of context dependencies: Commonly within compositions, ui-components depend on one another. For example, at a sequential composition of ui-components, subsequently executed components might require (parts of) the output of their predecessors. In addition, their mutually dependent activation must be controlled. According to concurrently activated ui-components, dependencies can be rather more sophisticated. To resolve these inter-component dependencies, ui-composition components must define specific context adaptation rules. Thereby, if an elementary component initiates a context change, then contexts of dependent ui-components are adapted accordingly.

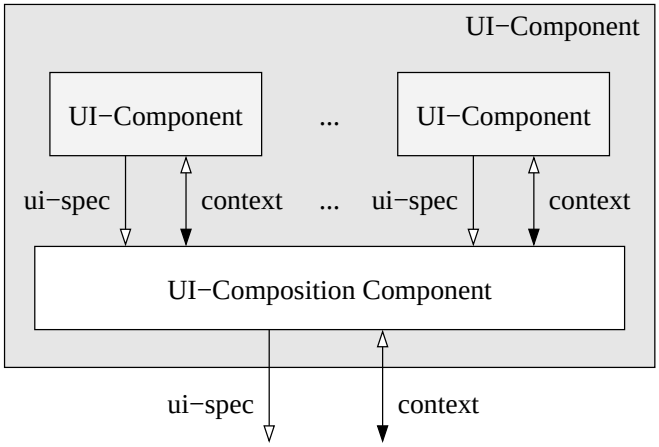


Figure 4.2: Composition of dependent ui-components

Figure 4.2 also indicates an essential feature of ui-composition components. The composition of a ui-composition component and according ui-components yields itself a ui-component. In other words, according to their external behavior, we do not distinguish elementary ui-components from composite ones. Therefore, the approach scales up to complex interaction specifications. If a composite interaction specification is provided, it can be used in turn for higher-level compositions.

Verification and simulation of interaction specifications is then based on verification and simulation of (possibly complex) ui-components. Besides an autonomous perspective, analysis of interaction specifications wrt. a targeted environment is applicable as well. For its support, we embed ui-components into an according environment which may be configured as required. As illustrated in Figure 4.3, we employ two components (i) ui-controller and (ii) user.

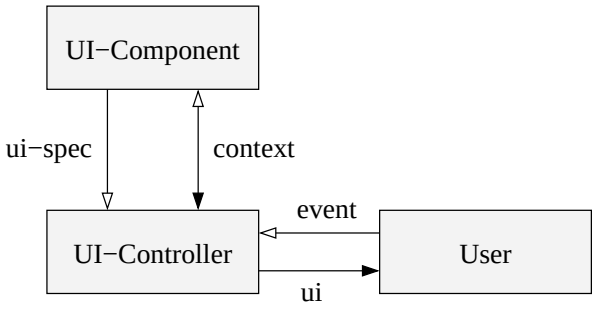


Figure 4.3: Embedding ui-components into an environment

An *ui-controller* realizes the interface between user and application. It can be compared to the known Model-View-Controller (MVC) architecture (cf., for example, [DFAB98]). While model and view are provided by (possibly complex) ui-components, the controller is specified by a distinguished ui-controller component. It realizes according responsibilities which basically comprise

1. receiving successive interactions specification from the ui-components,
2. conveying according system output and possible input alternatives to the user,
3. obtaining user events (corresponding to the input specification), and
4. notifying ui-components of received user events (and restart the loop at 1.).

Besides the controller, we may explicitly model the user by a specific *user component*. This component receives system output and activated events from the ui-controller. Afterwards it provides according user input. The abstraction of users by specific components is motivated by several advantages. It permits

- prototyping by realizing interaction with a concrete user. Simulators of CP nets provide facilities to interact with users during net simulation (cf. [Jen02]);
- simulation of predefined user scenarios. It is realized by initializing a user component by event sequences of interest;
- imposing behavioral peculiarities of specific user groups. For example, to simulate users that prefer a navigational interaction style, a corresponding user component may disregard other styles. Thereby, it emulates user's perception which focuses on navigation, but suppresses other alternatives.

4.2 UI-Components

As motivated in Section 4.1, we propose ui-components for the specification of user interaction. Through a common interface, they provide the opportunity of successive composition. Besides syntactical requirements wrt. their interfaces (cf. Figure 4.1 on page 74), we impose semantic restrictions onto ui-components which essentially include the specification of (i) assertions that must be provided by the environment, and (ii) properties that must be satisfied by ui-components.

Before we derive necessary restrictions in Section 4.4, we consider the realization of elementary ui-components more precisely. As illustrated in Figure 4.4, we propose to divide the internal structure of elementary ui-components into three essential parts: "UI-View", "Context", and "Context Transition Model". Although these parts rather represent conceptual constituents of ui-components, they may be defined as encapsulated (sub-)components themselves. In general, we may allow any internal structuring. However, the division proposed in Figure 4.4 yields a nice understanding of ui-components and simplifies their specification and composition.

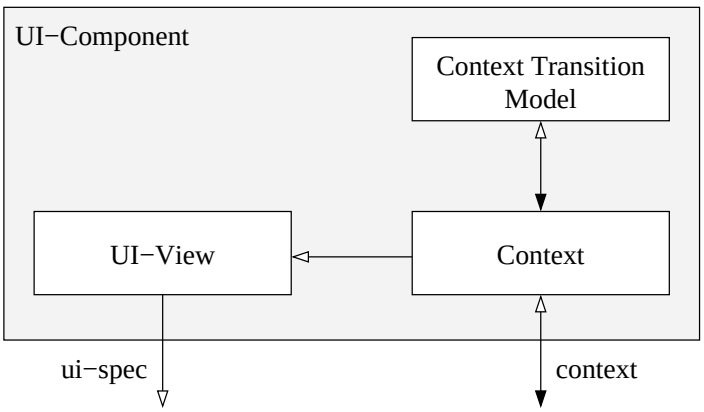


Figure 4.4: Internal structure of ui-components

As motivated in Section 4.1, ui-components provide (sub-)model and (sub-)view according to the Model-View-Controller abstraction. Thereby, ”UI-View” realizes the view and ”Context” together with the ”Context Transition Model” realize the underlying model. According to the model, the ”Context” component provides the static part of the model, i.e., the data. The ”Context Transition Model” provides the dynamic part of the model, i.e., data transformations. In the following, according responsibilities are delimited in detail.

UI-View

On external request, the ”UI-View” component computes the specification of the next dialog step, i.e., according system output together with enabled user events — possibly enriched by meta data, in particular, style information. It is realized by a function which requires a view onto the current context (corresponding to the current dialog situation) as an argument, and provides as output an interaction specification which can be interpreted by an according user interface controller.

Context

The context component manages the interaction context of a ui-component. Thereby, it reflects the current dialog situation wrt. the ui-component. For example, a ui-component that realizes a [Set-Based Navigation], may require (i) a list of elements, (ii) a pointer of the current list position, and (iii) a visible size, i.e., the number of list elements represented to the user at the same time. To access the context, the ”Context” component provides an interface for retrieval and manipulation.

To permit the resolution of context dependencies between ui-components, access can be provided to externally connected components as well. For example, if a [Set-Based Navigation] is composed with a ui-component that realizes a [Selectable Search Space], the underlying list of elements depends on the sub-space chosen by the user through [Selectable Search Space]. Thus, external adaptations of the element list may occur and must be imported.

Without according measures, manipulating shared data may cause any type of data inconsistencies. Therefore, we employ a database perspective and apply respective methods. Figure 4.5 motivates the fundamental idea to consider ”Context” components as *database views*, or more precisely, as database clients which operate on views. Consequently, we assume the existence of an integrated database system at the global level. Thereby, retrieval and manipulation requests initiated by a local ui-component are iteratively propagated to sub-sequent levels, until they are eventually processed (and materialized) by the database system. However, it is commonly not required to integrate the complete context of ui-components. Contexts which are independent from other ui-components can exclusively be managed locally. Therefore, contexts

can be declared as *private* or *public*. Private context is not visible (and, thus, not manipulatable) at higher levels. At a pragmatic perspective, context which is not involved within dependencies is declared as private. As a default, we might declare contexts as private. If later compositions impose context dependencies, a refinement in terms of re-declaring contexts as public can be applied. Note that the view perspective implies a minor notational conflict: the (public) contexts of lower-level ui-components correspond to higher-level views at a database perspective. However, if not stated differently, we will generally refer to 'higher-level' and 'lower-level' relations at the component perspective.

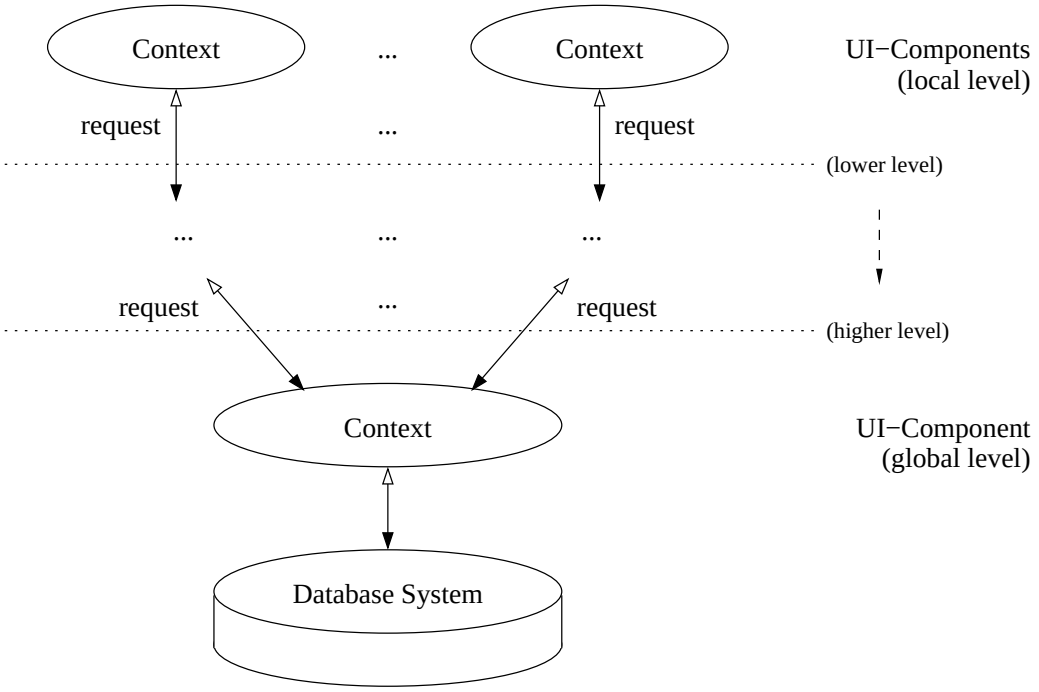


Figure 4.5: A view-based approach to ensure consistent context exchange

To avoid data inconsistencies as well as any kind of anomalies which are caused by concurrent access on shared data, we apply the notion of *transactions* known from the database area (cf., for example, [EN99, Bis95]). Context manipulations, for example, initiated by ECA rules are then executed within transactions. Thereby, so-called ACID properties, i.e., atomicity, consistency, isolation, and durability are ensured.

Context Transition Model

The "Context Transition Model" reflects the dynamic behavior of a ui-component. It essentially specifies the transition between successive dialog situations. It is realized

by initiating manipulation requests to the context. There exist two types of events which may initiate a transition of dialog situations:

- An event provided by the current dialog specification is initiated by a user. For example, a ”next” event can be initiated by a user according to a ui-component that realizes a [Set-Based Navigation]. It entails a context adaptation request ”Increase the list pointer by the value of visible size”.
- In case of context dependencies, context manipulation of one ui-component may require subsequent context adaptation of dependent ui-components. For example, if a different sub-space is chosen by a user through a [Selectable Search Space], the element list of dependent ui-component that realizes [Set-Based Navigation] must be adapted accordingly.

As motivated in Section 2.3, a natural opportunity to specify context transitions are provided by the method of *Event-Condition-Action rules* (ECA rules). We utilize them as follows:

Events correspond to context changes,

Conditions correspond to boolean-valued functions on the context, and

Actions correspond to context manipulation requests.

Note that we represent user events by contexts as well. Thereby, initiated user events entail context changes which will activate associated rules. To avoid inconsistencies, the ”UI-View” component is blocked during the execution of the rule system. Otherwise, it may obtain inconsistent context, since some dependencies might not yet been resolved completely. The termination of the rule system is considered at a global level. More precisely, the rule system terminates, if no rule of any of the ui-components is activated. Note that rule systems can generally be characterized by properties as termination, confluence, and effect preservation. Since rule systems do not guarantee these properties at the general case, according restrictions must be applied. For a discussion about these issues, we refer to Section 4.7.

In the following, we will outline an example. We consider a possible formalization of interaction pattern [Set-Based Navigation]. An according ui-component named *’List Scrolling’* is specified as follows.

An Exemplary UI-Component: ’List Scrolling’

The realization of ui-component ’List Scrolling’ is based on the specification of its three sub-components ”Context”, ”UI-View”, and ”Context Transition Model”. An according graphical abstraction which outlines the essential elements is provided in Figure 4.6.

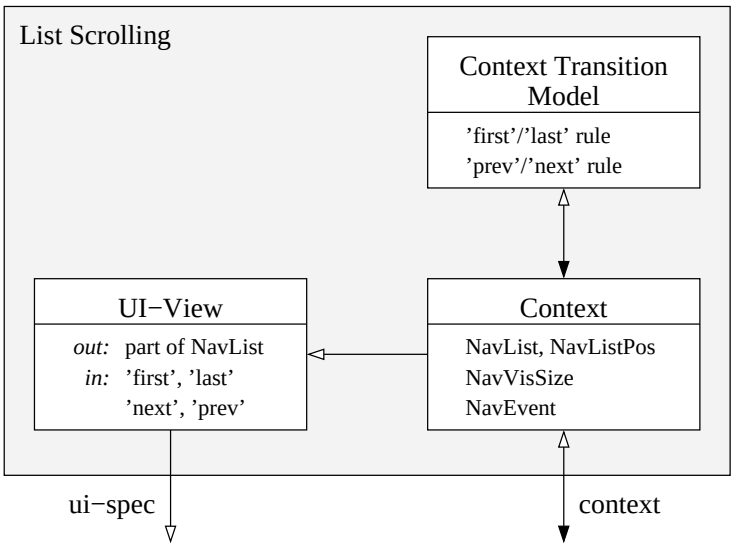


Figure 4.6: UI-Component 'List Scrolling'

Context:

NavList: represents the underlying list of elements which may be browsed.

NavListPos: represents the current list position (starting at 1).

NavVisSize: represents the number of list elements presented to a user at the same time.

As we represent events by context, we employ:

NavEvent: represents the initiated user event according to this ui-component. It comprises events: 'next', 'prev', 'first', 'last'.

UI-View: At a basic level, it provides a plain interaction specification without stylistic information. The specification of (i) the output to the user and (ii) the activated user events are generated as follows:

Output: Sublist of *NavList* starting at position *NavListPos* and comprising *NavVisSize* elements. If the sublist exceeds the end of *NavList*, it is shortened accordingly.

Activated events: Depending on the context, user events 'next', 'prev', 'first', 'last' are activated:

'next', 'last': are activated, if

$$NavListPos + NavVisSize \leq length(NavList)\,,$$

where *length*() provides the number of elements of a list. Thereby, event 'next' corresponds to navigating to the succeeding part of the list, and event 'last' corresponds to jumping to the last part of the list — as specified by the context transition model.

'prev', 'first': are activated, if

$$NavListPos > 1\,.$$

Thereby, event 'prev' corresponds to navigating to the preceding part of the list, and event 'first' corresponds to jumping to the first part of the list.

Context Transition Model: The context is adapted depending on the user event initiated. According rules essentially adjust the list pointer wrt. the user event. They are specified as follows:

'first' rule: Jump to the beginning.

```
ON updated(NavEvent)
IF NavEvent == 'first'
DO update(NavListPos) by 1
```

'last' rule: Jump to the end.

```
ON updated(NavEvent)
IF NavEvent == 'last'
DO update(NavListPos) by length(NavList) - NavVisSize +1
```

'next' rule: Navigate forwards.

```
ON updated(NavEvent)
IF ( (NavEvent == 'next') &&
   (NavListPos + NavVisSize  <=  length(NavList) )
DO update(NavListPos) by NavListPos + NavVisSize
```

'prev' rule: Navigate backwards.

```
ON updated(NavEvent)
IF ( (NavEvent == 'prev') && (NavListPos > 1) )
DO {
   IF ( NavListPos > NavVisSize )
     update(NavListPos) by NavListPos - NavVisSize
```

```
ELSE
  update(NavListPos) by 1
}
```

The chosen pseudo-code syntax to represent context transition rules combines the ECA style with that of the programming language C. Its intention is to provide an intuition of the behavior rather than a specification. The formal specification is based on interaction nets. However, as discussed in Section 4.4.2, a similarly convenient, but formal specification language might be provided to generate according interaction nets. According to the syntax deployed above, we represent ECA rules by the following general frame:

```
ON <event>
IF <condition>
DO <action>
```

Events (propagated by context changes) are denoted by boolean-valued expressions

```
'updated(<contextName>)'.
```

They evaluate to *true* or *false* depending on whether the context identified by its name was manipulated or not. A retrieval of context values is simply denoted by the name of the context. A context update is denoted by

```
'update (<contextName>) by <expression>'
```

which corresponds to an update of the identified context by the evaluation of the specified expression.

UI-Component 'List Scrolling': An Open Perspective

The proposed realization of ui-component 'List Scrolling' employs an autonomous perspective. It assumes that its context is locally used only, and may not be adapted by other ui-components. However, to enable dependencies between ui-components, we must provide a more liberal (or open) perspective. It is achieved by (i) declaring contexts as 'public', and (ii) extending the context transition model by according adaptation rules. For example, to allow other components to update or completely replace the underlying navigation list, context *NavList* will be declared as public. In addition, the following adaptation rule must be included into the context transition model. It resets the list pointer, since an update of *NavList* might cause the current list pointer to become invalid, i.e., to point outside the modified list. Intuitively, such adaptations can be understood as 'repairing actions'. They are applied to re-establish a consistent state of the local context.

'List' rule: Reset list position.

```
ON updated(NavList)
DO update(NavListPos) by 1
```

Note that we suppress the condition part of ECA rules, if it constantly evaluates to *true*. We will apply this open perspective in the next section, when we compose 'List Scrolling' with a ui-component which represents a [Selectable Search Space].

4.3 UI-Composition Components

4.3.1 Composition of UI-Components

As illustrated in Section 4.1, we employ ui-composition components to derive complex ui-components from elementary ones. They specify dependencies between elementary ui-components. It includes (i) the composition of the individual dialog step specifications and (ii) the resolution of context dependencies. Figure 4.7 proposes an internal structure which basically associates these tasks with according sub-components.

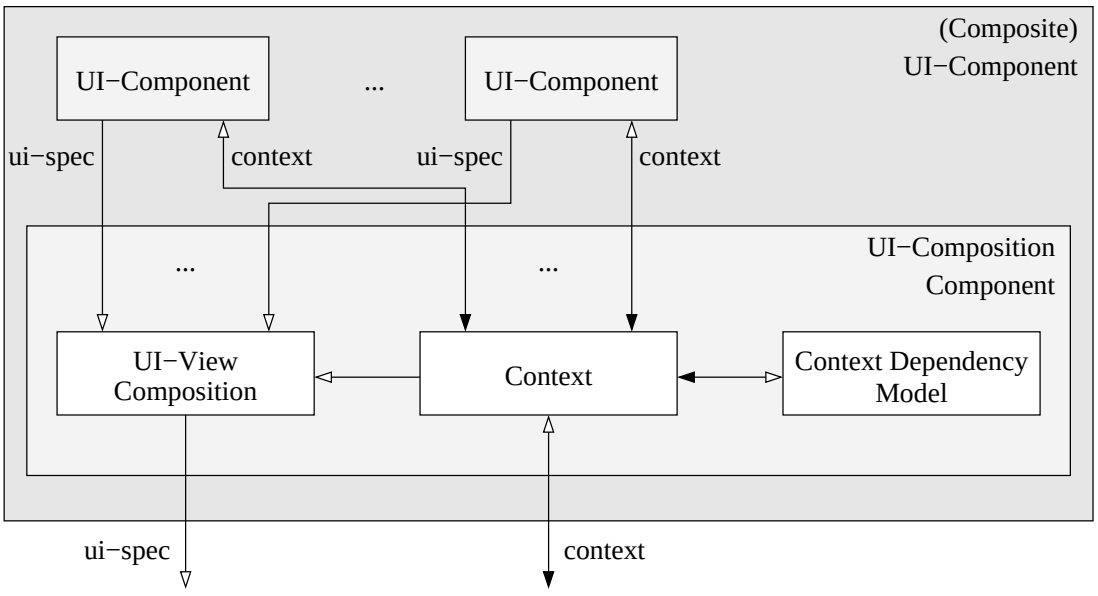


Figure 4.7: Internal structure of ui-composition components

The responsibilities of the three sub-components "UI-View Composition", "Context", and "Context Dependency Model" can be compared to that of sub-components "UI-View", "Context", and "Context Transition Model" of elementary ui-components. In

addition, they have to provide the integration of attached ui-components. As motivated in Section 4.1, a composite ui-component provides the model and the view according to the Model-View-Controller abstraction. Thereby, the view is realized by the sub-views of attached ui-components together with their integration realized by sub-component ”UI-View Composition”. Accordingly, the model is realized by the sub-models of attached ui-components together with their integration realized by sub-components ”Context” and ”Context Dependency Model”. In the following, according responsibilities are delimited in detail.

UI-View Composition

The responsibility of this component can be compared with the ”UI-View” component of elementary ui-components. It computes the specification of the next dialog step, i.e., according system output together with enabled user events — possibly enriched by style information. In contrast to ”UI-View”, it is realized by (i) receiving ui-specifications of attached (lower-level) ui-components and subsequently (ii) composing an integrated ui-specification. Thereby, the composition may evaluate style information provided by elementary ui-specifications as well as context information retrieved from the context component.

Context

The context component manages the interaction context of the composite ui-component. At a non-integrated scenario, it basically comprises the union of the public contexts of attached ui-components which may further be extended, if required. However, there may also occur situations where

(i) contexts of different ui-components correspond to one another, and

(ii) equally named contexts of different ui-components do not correspond to one another.

While in case (i) both contexts should be merged at the higher level, in case (ii) both contexts should be separated. Note that these alternatives represent one opportunity to specify context dependencies. A more general opportunity is provided through the context dependency model. Note further that at a general perspective, context integration at this level corresponds to the *view integration problem* known from the database area (cf., for example, [BCN92] and Figure 4.5 (on page 79)).

Thereby, the context reflects the current dialog situation of the composite ui-component. It provides retrieval and manipulation interfaces (i) to attached (lower-level) ui-components and (ii) to an (higher-level) ui-component. Note that according to the view perspective in Figure 4.5, we assume retrieval and manipulation requests as *directed*

requests. More precisely, retrieval and manipulation requests are initiated by lower-level ui-components and passed to higher-level ui-components for processing.

Context Dependency Model

The ”Context Dependency Model” specifies context dependencies between attached (lower-level) ui-components. Analogous to elementary ui-components, they can be expressed in terms of ECA rules. Together with the lower-level context transition models, it reflects the dynamic behavior of the composite ui-component. In other words, it essentially specifies the transition between successive dialog situations wrt. the composite ui-component. Therefore, we may denote the union of the (lower-level) context transition models together with the ”Context Dependency Model” as ”Context Transition Model” of the composite ui-component.

Together, there exist three alternatives to dependency resolution:

Integration of synonymous concepts: If attached ui-components denote synonymous concepts by different names, their integration is specified by the context component.

Separation of homonymous concepts: If attached ui-components denote distinguished concepts by the same name, their integration is specified by the context component as well.

Complex dependencies: In the case of complex dependencies between attached ui-components, dependency rules are specified by the context dependency model.

Analogous to elementary, ui-components, ”UI-View Composition” is blocked during the execution of the rule system to avoid inconsistent views.

In the following, we outline the realization of an exemplary composition. We consider a composition of ui-component ’List Scrolling’ introduced in Section 4.2 which realizes a [Set-Based Navigation] and a ui-component ’Category Selection’ which realizes a [Selectable Search Space]. To specify their composition, we will first propose a possible realization of ui-component ’Category Selection’:

UI-Component: ’Category Selection’

The realization of its according sub-components is described in the following. We assume that the underlying ’search space’ is represented by a base set of elements. Sub-space division is then represented by a function which given a selected category, it provides an associated subset of the base set. Figure 4.8 provides an according graphical abstraction.

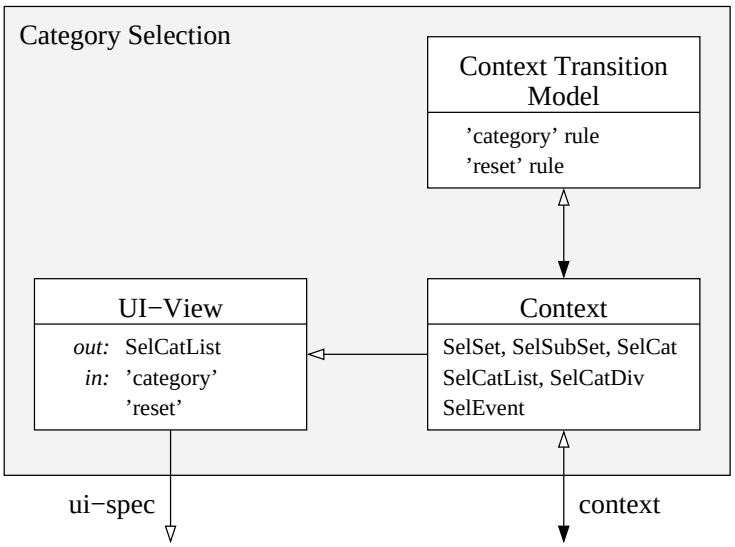


Figure 4.8: UI-Component 'Category Selection'

Context:

SelSet: represents the underlying set of elements.

SelCatList: represents a non-empty list of available categories.

SelCat: represents the category selected currently, or constant 'null' if no category is selected.

SelSubSet: represents the subset associated with the currently selected category *SelCat*.

SelCatDiv: determines the division of *SelSet* into subsets according to categories in *SelCatList*. It is represented by a list of pairs of a category and its associated subset.

As we represent events by context, we employ:

SelEvent: represents the initiated user event according to this ui-component. It comprises elements of *SelCatList* and a 'reset' event.

UI-View: At a basic level, it provides a plain interaction specification without stylistic information. The specification of (i) the output to the user and (ii) the activated user events are generated as follows:

Output: List of *SelCatList*.

Activated events: Each element of *SelCatList* is activated as user event with the exception of the currently selected category *SelCat*. Thereby, the selection of a category corresponds to selecting an associated subset *SelSubSet*. If a sub-space is currently selected, a specific 'reset' event is provided additionally which permits to reset user's scope to the initial space.

Context Transition Model: The context is adapted depending on the user event initiated. According rules essentially adjust the category selected *SelCat* and its associated subset *SelSubSet*:

'category' rule: Select sub-space.

```
ON updated(SelEvent)
IF ( SelEvent in SelCatList )
DO {
   update(SelCat) by SelEvent ;
   update(SelSubSet) by SelCatDiv[SelEvent]
   }
```

where '*in*' denotes element containment.

'reset' rule: Select original space.

```
ON updated(SelEvent)
IF ( SelEvent == 'reset' )
DO {
   update(SelCat) by 'null' ;
   update(SelSubSet) by SelSet
   }
```

A dependent composition between 'List Scrolling' and 'Category Selection' is realized by the following ui-composition component:

UI-Composition component: 'Category Selection' + 'List Scrolling'

The intention of this composition is to provide a minimal interactive catalog to users. They may browse the complete catalog by initiating activities provided by 'List Scrolling' or may decide for browsing specific sub-spaces of the catalog. There exists a (directed) dependency from 'Category Selection' to 'List Scrolling'. If a sub-space is selected (or deselected) by the user wrt. 'Category Selection', the according element list provided by 'List Scrolling' must be adapted. This dependency is specified by a context dependency rule (as defined below). The ui-composition component is then realized as follows. Figure 4.9 provides a corresponding graphical abstraction.

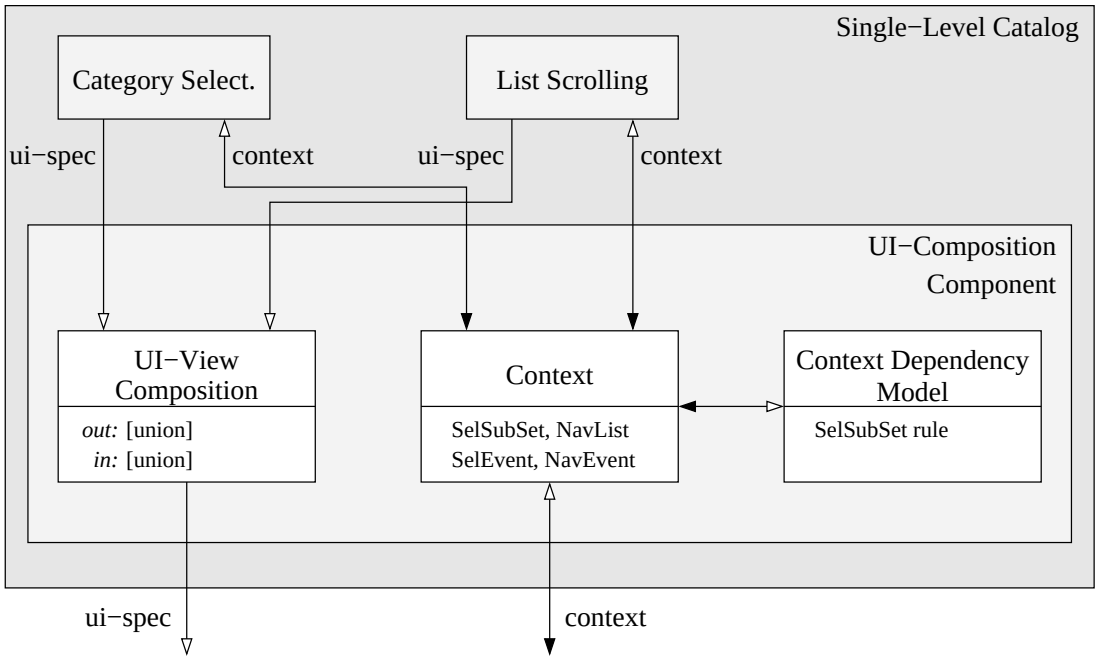


Figure 4.9: A concrete ui-composition component

Context: *SelSubSet* and *NavList* are assumed to be declared as public by ui-components 'Category Selection' and 'List Scrolling' respectively. Since events are initiated by the ui-controller, *SelEvent* and *NavEvent* must be declared as public as well. A further extension of the context is not required.

UI-View Composition: At a basic level, interaction composition is provided by the union of (i) the user outputs and (ii) the activated events of both ui-components.

Context Dependency Model: The dependency is specified by the following rule. It basically copies the content of *SelSubSet* to *NavList*.

'SelSubSet' rule: Publish altered sub-space.

```
ON updated(SelSubSet)
DO update(NavList) by list(SelSubSet)
```

where '$list()$' denotes type conversion of sets to lists (based on an arbitrary ordering).

The composition of the ui-composition component together with ui-components 'Category Selection' and 'List Scrolling' then provides a composite ui-component which corresponds to a simplified catalog interaction. We call this component 'Single-Level Catalog'.

4.3.2 Composition Patterns

The exemplary composition represented in Section 4.3.1 deploys a ui-composition component which exclusively specifies the dependency between a particular pair of ui-components. In general, it is not always required to introduce special purpose compositions. If a composition reflects a common and recurring policy, it seems promising to provide ui-composition components which can generically be applied to different ui-components. We refer to such common policies as *composition patterns*. A characteristic example of composition patterns are policies governing the control flow of dialogs. A frequent policy is a sequential composition of dialog structures. For example, (i) after the details of a journey have been confirmed, dialog steps wrt. the actual payment are invoked, or (ii) after a student registered for a course, materials may be downloaded, and messages may be posted at a course-related forum.

We will consider the example of sequential composition in the following to illustrate how composition patterns can generically be provided by ui-composition components. Note that there exists a substantial distinction between sequential composition of dialog structures and sequential composition of components or interaction nets. While the later rather corresponds to a subsequent transformation of data elements (comparable to 'pipes' at command processing), the former concerns the flow of control. We first consider the case that sequential composition coincides with the structure of ui-components. More precisely, interactive facilities of one component are activated first. As soon as a sequence of user interaction reaches a final state wrt. this ui-component, the second ui-component is activated. We propose two alternative realizations of sequential composition distinguished by the allocation of responsibilities.

Sequential composition (by centralized responsibility)

The responsibility of activation and deactivation of ui-components is completely realized by a ui-composition component. It basically comprises two tasks. (1) For each associated ui-component X, a private context 'activeX' is maintained which represents their current status of activation, i.e., the context indicates which ui-component is currently activated. If an event indicates a completion of the current sub-dialog, then activation states are adapted, such that the next ui-component becomes activated. It is realized by specifying according context dependency rules. (2) The "UI-View Composition" component reflects the current activation by filtering dialog specifications received from associated ui-components. If an associated ui-component is currently inactive, then its provided dialog specification is disregarded by the view composition. A dialog specification received from an currently active ui-component will be provided to the next higher level only.

As an advantage, this approach is generally applicable. However, it possesses some drawbacks. Firstly, dialog specifications of inactive ui-components are computed, although they are disregarded later on. As we currently do not focus on aspects of

performance, we less emphasize this issue. Secondly, the responsibility of recognizing the completion of a sub-dialog wrt. an associated ui-component is realized by the ui-composition component. However, as the semantics of sub-dialogs is specified by a particular ui-component, the ui-component itself should commonly be responsible for recognizing its completion. For example, a ui-component that realizes interactive scenarios of payment "knows" by which event and at which dialog situation payment is completed.

Sequential composition (by localized responsibility)

A refined version that remedies above drawbacks then shifts responsibilities to the associated ui-components. Thereby, each ui-component *X* maintains a particular (and public) context 'finishedX' which indicates its completion. In addition, a context 'activeX' is maintained for each associated ui-component as introduced above. While contexts 'finishedX' are updated by the respective ui-component *X*, contexts 'activeX' are updated by the ui-composition component. Thereby, ui-components locally respect context 'activeX', i.e., they suppress the generation of a dialog specification, as long as 'activeX' indicates an inactive status. An ECA rule of the ui-composition component which specifies a sequential composition of two ui-components *A* and *B* reads as follows:

```
ON updated(finishedA)
DO {
   update(activeA) by 'false' ;
   update(activeB) by 'true'
   }
```

By the introduced realization, sequential composition of ui-components is provided in terms of generic ui-composition components. Their correct behavior can be verified through respective specifications on the basis of interaction nets. Note that the localized alternative imposes an assertion onto associated ui-components. They have to maintain context 'finishedX' and to respect context 'activeX'. It implies the following design obligation. For a ui-component which intends to be employed in sequential scenarios, this property of "sequentialization" must be provided.

The introduced realization of sequential composition motivates natural generalizations. Firstly, any number of involved ui-components can be chosen. Then, an according rule of activation adaption must be specified for each 'finishedX' event Secondly, generalized alternatives of dialog flow are realizable. By specifying according activations, any kinds of dialog flow are possible as, for example, cyclic sub-structures. Thirdly, it may occur that the activation does not concern single ui-components, but sub-sets of ui-components. For example, we might employ an interaction history throughout a sequential composition of ui-components. Therefore, at each dialog situation at least

two ui-components will be active in parallel: (i) the interaction history, and (ii) the ui-component of the sequential composition. It is straightforward realized by according activation adaptation rules. Thereby, the activation context of the interaction history remains active throughout the complete dialog.

General composition patterns

The notion of composition patterns naturally generalizes from the scope of ui-components to components in general. We refer to according components which realize generic composition policies as *composition components*. They commonly play the role of a mediator between connected components. A typical example are service request queues. Services provided by certain components are commonly requested by more than one client. For example, manipulation requests to a database system are used by several applications and clients. Similarly, at a component-based approach, database requests will be required by several components. However, we cannot simply connect several components through a single service interface, because of the definition of component composition (cf. Definition 3.5 on page 33). While the component approach of Broy et. al. [BS01] permits such a ”1 channel to many channels” composition, its dynamic semantics is not the one required for this purpose. For example, assume that an output channel of a service provider component is connected to input channels of several client components. By Definition 3.5, it corresponds to providing one and the same copy of the output to every connected client. Obviously, this is not the required behavior. We rather expect that the output is provided to the selected client who initiated an according service request.

An elegant and flexible solution to this problem is to deploy a generic composition component which encapsulates the behavior of a service queue. It is flexible, since it permits to specify the desired policy of a queue. For example, a particular queue component might specify the standard behavior of a ”first come, first serve” policy, while another queue might prioritize requests. Figure 4.10 demonstrates the deployment of a queue component by a black-box interaction net. Interaction net specifications of different queue policies will be provided in Section 4.4.

4.3.3 Refinement and Adaptation

At a practical point of view, it might be impossible to provide a repository of ui-components which satisfies all current and future requirements. A designer will often be confronted with the situation that there exist ui-components which meet the design requirements approximately only. In this case, a refinement or an adaptation of a selected ui-component must be established. Besides the opportunity of adapting the specification of the ui-component directly, a so-called wrapper approach may relieve the adaptation task significantly. As illustrated in Figure 4.11, a wrapper is a component which provides a (dynamic) view onto an underlying ui-component. We

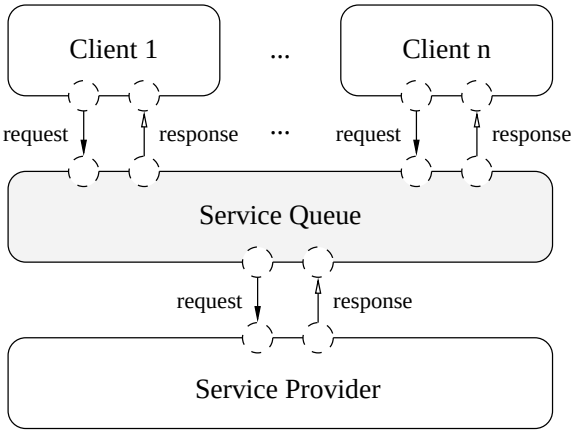


Figure 4.10: A composition component providing a service queue

denote wrappers used wrt. ui-components as *ui-wrappers*. The figure also indicates that the composition of a ui-wrapper and its underlying ui-component represents a ui-component in turn. Thus, analogous to the composition of ui-components, the wrapper approach scales up to adapted interaction specifications. In other words, adaptation may be applied iteratively — therewith providing different versions of ui-components.

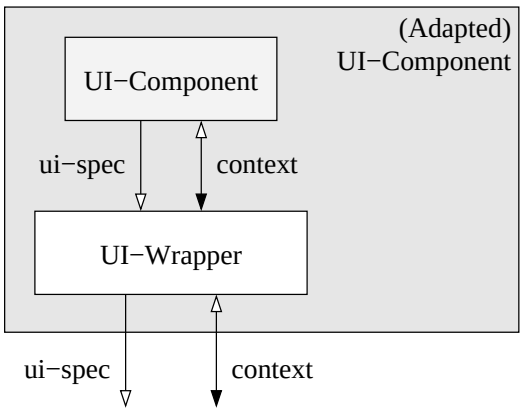


Figure 4.11: Refining and extending ui-components by wrappers

As illustrated in Figure 4.12, ui-wrappers can be realized by *unary* ui-composition components. They permit adaptation of the view as well as of the underlying model specification. We slightly rephrased its sub-components (i) 'UI-View Composition' to 'UI-View Adaptation', since there is only one argument view, and (ii) 'Context Dependency Model' to 'Context Adaptation Model', since there is only one underlying model which is adapted through this component. Characteristic adaptations are, for example:

- extending the dialog specification by additional output and user events,
- enriching the dialog specification by style information,
- adapting the behavior by according context rules, or
- extending and adapting the context.

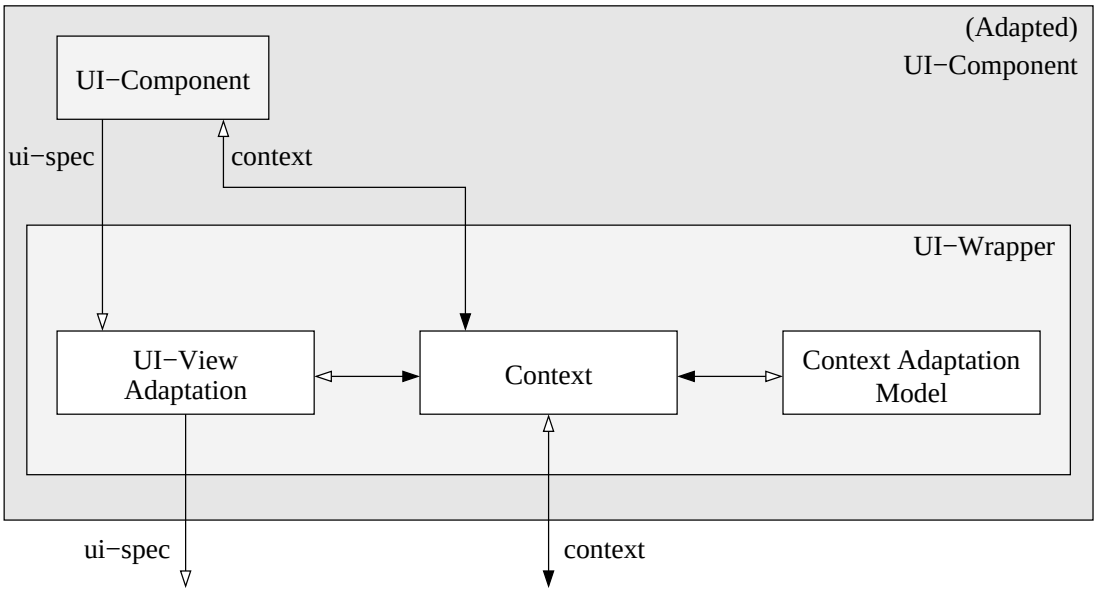


Figure 4.12: Realization of ui-wrappers by unary ui-composition components

UI-Wrapper: 'Circular Navigation'

In the following, we illustrate an exemplary situations where the use of ui-wrappers is beneficial. We consider an adapted requirement according to ui-component 'Set-Based Navigation':

> "User events 'next' and 'prev' should be enabled unconditionally. If the end of the list is presented to the user, a subsequently initiated 'next' event entails a jump to the beginning of the list. If the beginning of the list is presented, a subsequent 'prev' event entails a jump to the end of the list."

An according ui-wrapper which realizes this adaptation wrt. ui-component 'List Navigation' can be specified as follows. A graphical illustration is provided in Figure 4.13.

Context: corresponds to the context of ui-component 'List Navigation'. A further context extension is not required.

UI-View Adaptation:

Output: The output specification of attached ui-component 'List Scrolling' is provided unchanged.

Activated events: The input specification of attached ui-component 'List Scrolling' is extended by events 'next' and 'prev', if they do not occur.

Context Adaptation Model: According to events 'next' and 'prev' the list pointer is adapted. It is realized by initiating corresponding events 'first' or 'last' which will be processed subsequently by the elementary ui-component 'List Scrolling'.

'next' adaptation rule: If end of list is reached, jump to the beginning.

```
ON updated(NavEvent)
IF ( (NavEvent == 'next') &&
     (NavListPos + NavVisSize > length(NavList)) )
DO update(NavEvent) by 'first'
```

'prev' adaptation rule: If beginning of list is reached, jump to the end.

```
ON updated(NavEvent)
IF ( (NavEvent == 'prev') && (NavListPos = 1) )
DO update(NavEvent) by 'last'
```

As indicated by Figure 4.13, the composition of the ui-wrapper and ui-component 'List Scrolling' provides an adapted ui-component we call 'Circular List Navigation'.

User adaptation

Besides a rather task-oriented adaptation of dialog structures, ui-wrappers can be utilized to specify user adaptation. An according ui-wrapper represents a user profile (including user preferences) by introducing necessary contexts. User adaptation is then specified at two dimensions: by context adaptation and by user interface adaptation. Concerning context adaptation, according adaptation rules must be specified. As a simple example, consider a boolean option of the user preferences which determines whether to apply circular navigation or not. Firstly, an according boolean-valued context 'Circular' will be introduced which reflects the setting of the option. Secondly, rules corresponding to above 'next' and 'prev' adaptation rules are introduced. Thereby, their condition parts are extended by the evaluation of option 'Circular'. The according 'next' adaptation rule then reads as follows:

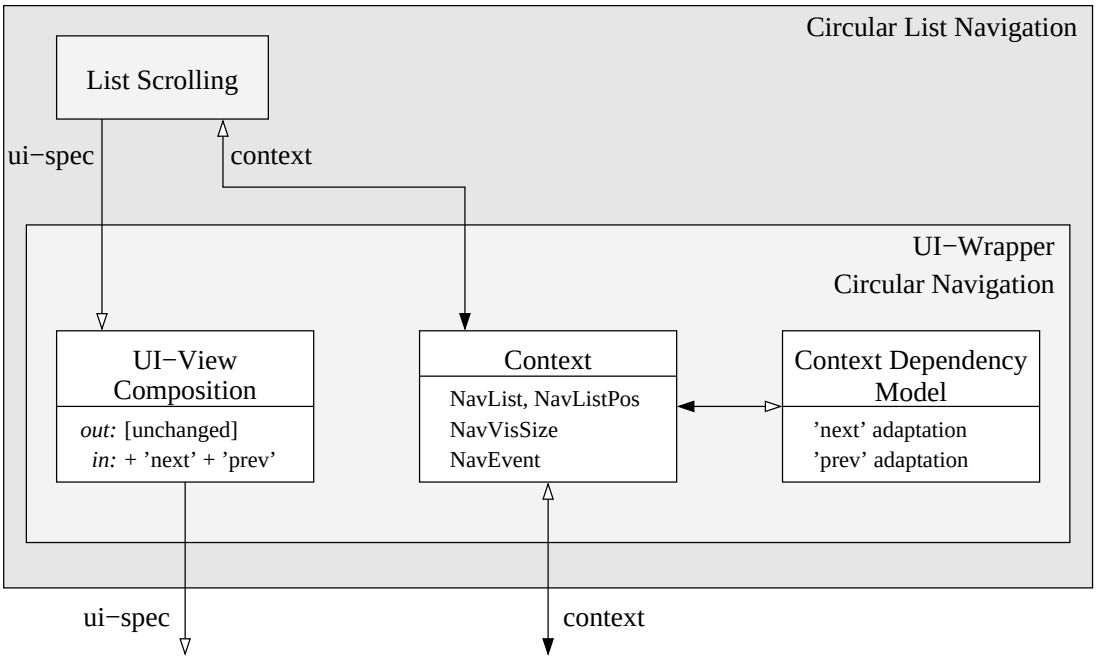


Figure 4.13: a ui-wrapper that enables circular navigation

'circular(next)' adaptation rule: If 'circular' is set and end of list is reached, jump to the beginning.

```
ON updated(NavEvent)
IF ( (Circular == 'true') &&
     (NavEvent == 'next') &&
     (NavListPos + NavVisSize > length(NavList)) )
DO update(NavEvent) by 'first'
```

While context adaptation mainly permits the adaptation of dialog structures, "UI-View Adaptation" mainly focuses on the representation. Thereby, it enables to include style information as well as to evaluate style information according to the user profile.

Note that the proposed opportunity of user adaptation also provides a contribution wrt. consistency. More precisely, if user adaptation is correctly specified at a possibly global level, its context adaptation affects all (lower-level) ui-components. In other words, user adaptation can be specified globally. Therefore, contradicting adaptations which occur if specification of user adaptation is distributed over the application can be prevented.

4.4 Realization Based on Interaction Nets

In this section, we introduce a realization alternative based on interaction nets according to the component architecture outlined in Section 4.1 and further elaborated in Sections 4.2 and 4.3. The specification of a concrete realization essentially intends to provide a 'proof of concept'. In particular, we neglect aspects of performance at the moment. For example, we employ a simplified specification for the provision of a transaction service which can be understood as a "light-weight" transaction model. It basically performs transactions sequentially. However, at a semantic level, it corresponds to transaction services that permit concurrent executions.

At the perspective of realization, we barely distinguish between components "UI-View" and "UI-View Composition" as well as between components "Context Transition Model" and "Context Adaptation Model". They are treated rather analogously. Discussion about minor distinctions are appended to the end of the respective sections. Therefore, a formal specification of "UI-Components" based on interaction nets is basically provided by the three sub-components "Context", "Context Transition Model", and "UI-View" (cf. Figure 4.4 on page 77). Their specifications are introduced subsequently at the following sections.

4.4.1 Context

As motivated in Section 4.2, the context component shall provide access to shared data. For simplicity, we encapsulate each shared data by a distinguished component which may be called by an associated context name. For example, list *NavList* and integer *NavVisSize* occurring at the context of ui-component 'List Scrolling' can be encapsulated by distinguished components called 'NavList' and 'NavVisSize'. Through an according interface, shared data components provide access to the data. As proposed in Section 4.2, we permit to declare data as private or public. While private data provides an opportunity for encapsulation, public data is required for dependency resolution. According to their realization, we apply a database perspective previously motivated at Figure 4.5 on page 79. Thereby, *public data* corresponds to a view onto persistent data at a higher level. In other words, requests to public data are simply delegated to the next higher level. Therefore, we also refer to them as *view*. *Private data* corresponds to the actual (persistent) storage of the data. Therefore, we also refer to them as *persistent data*. According to the view perspective, data can be declared as persistent at one selected level only. According to database systems, each data is made persistent at the database level — which corresponds to the global level (cf. Figure 4.5). However, to permit encapsulation at each level, we enable a more liberal approach. Data may be declared as private/persistent at any level. Thereby, private data is accessible at the same and (possibly) lower levels only.

Figure 4.14 illustrates an example. It reflects a part of the context realization of composite ui-component 'Single-Level Catalog' represented in Figure 4.9 on page 89. For

readability, sub-components 'UI-View' and 'Context Transition Model' are omitted from the figure. We emphasized sub-components that provide persistent data by thick boxes. Corresponding to arrows styles, Figure 4.9 adheres to the convention stipulated previously. While filled arrows denote the direction of requests, hollow arrows denote corresponding responses. Thereby, requests represented in the figure primarily denote retrieval and manipulation requests.

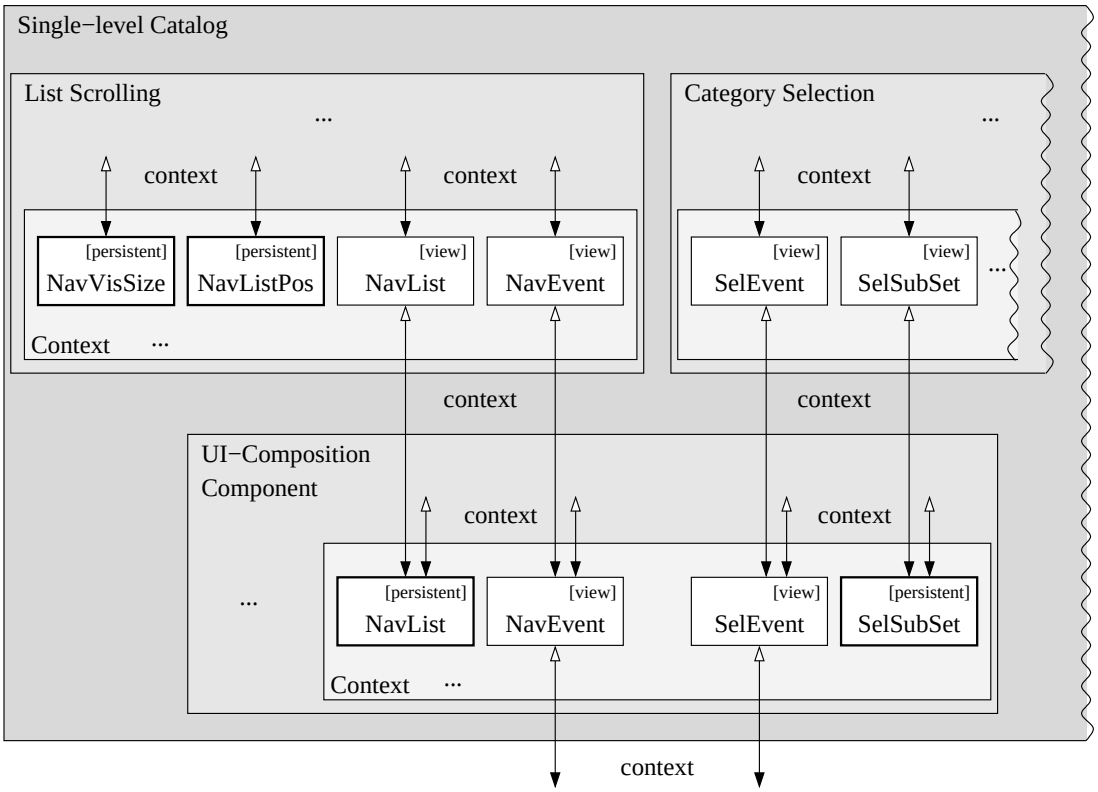


Figure 4.14: Context realization wrt. 'Single-Level Catalog'

Since a direct manipulation of shared data may cause inconsistencies, we include an elementary transaction management. It is realized by a sub-component called *'TA Service'* ('Transaction Service') which is employed at each context component. Figure 4.15 outlines the generic structure of a context component at the global level. There, all shared data is persistent.

At an intermediate level, data may be either persistent or realized by a view. The according transaction service at an intermediate level can be understood as a view as well — a view onto the global transaction service. It delegates requests to the next higher level. In contrast to shared data, we only deploy a single "persistent" transaction service at the global level. Figure 4.16 outlines the generic structure of a context component at a local level. The represented declaration of data D_1 as view and data D_n as persistent should be understood as an example case.

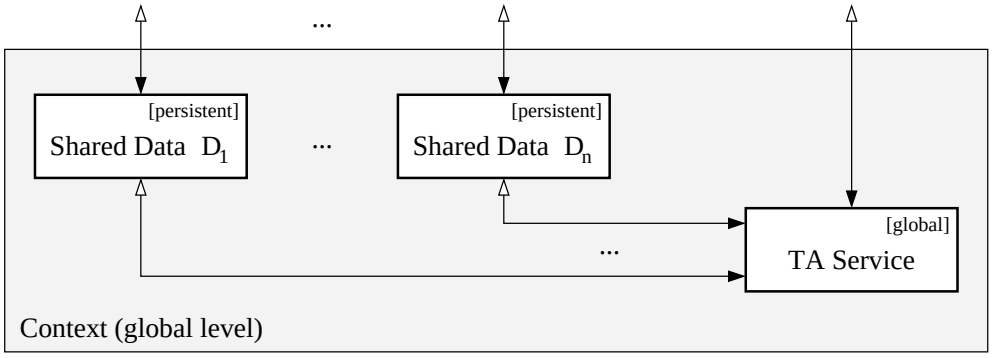


Figure 4.15: Generic structure of a context component (at global level)

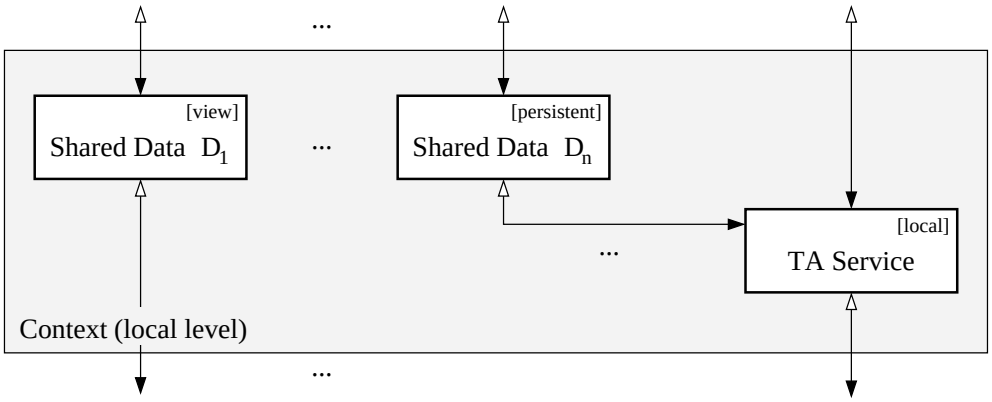


Figure 4.16: Generic structure of a context component (at a local level)

Transaction Service

We start by specifying the transaction service at the global level. Its main intention is to prevent data inconsistencies which may occur during concurrent executions of ECA rules (as defined by the context transition model). A straightforward solution is to enforce a sequential execution of transactions and, consequently, ECA rules as well. At a simplified transaction management, we will also neglect features as rollback or recovery. This transaction service will be sufficient to fulfill our requirements. However, it may be replaced by a more advanced system, if required.

Figure 4.17 represents the transaction service component ”TA Service” as a black-box interaction net. It provides interfaces for the following services:

Begin Transaction: This service is requested by a client, if it wishes to start a transaction. The provider waits until no more transaction is running currently. Then, it increments transaction time and returns a new transaction identifier *tid* to the

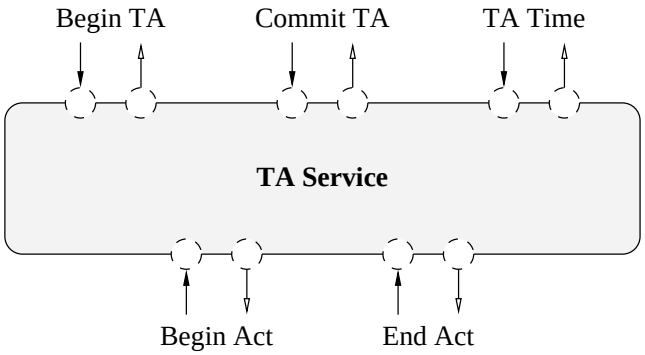


Figure 4.17: Interfaces of the transaction service component

client. By the unique transaction identifier, a client will be permitted to initiate retrieval and manipulation actions on shared data. The service is mainly used by ECA rules.

Commit Transaction: This service is requested by a client, if it wishes to commit a transaction. The provider invalidates the transaction identifier and permits new transactions to be initiated.

Begin of Action: By this service, the execution of a data retrieval or manipulation action is requested by a client. The provider verifies the transaction identifier provided by the client. Only if it corresponds to the identifier *tid* of the currently running transaction, a positive response is returned. Requests for 'Commit Transaction' are postponed, until the end of an action will be declared. This service is mainly used by services 'Retrieve' and 'Update' provided by shared data.

End of Action: This service declares an action to be finished. Afterwards, further actions (including transaction commit) may be requested for execution.

Transaction Time: By this service, the current transaction time is provided to a client. Its realization can be understood as a 'wake up' call initiated by the transaction service. More precisely, the client provides a transaction time — commonly the next transaction time. Afterwards, the current transaction time is returned to the client, if two conditions become true: (i) there is no transaction currently running, and (ii) the current transaction time equals or exceeds the transaction time provided by the client. The second condition is motivated by the following situation. A client who needs to be informed about an advance of transaction time would have to initiate requests to the "TA Service" within a polling loop to finally perceive an increment. Through condition (ii), a client only needs to initiate a single request. It will be called back automatically by the transaction service, if transaction time increased.

This service is mainly used by a service called 'Updated' provided by shared data. Thereby, service 'Updated' is essential to realize the 'detection' of events (concerning ECA rules). A detailed explanation is postponed to the specification section of shared data.

An interaction net specification of the transaction service is represented in Figure 4.18. For readability, we additionally emphasized initial place markings by dots within places.

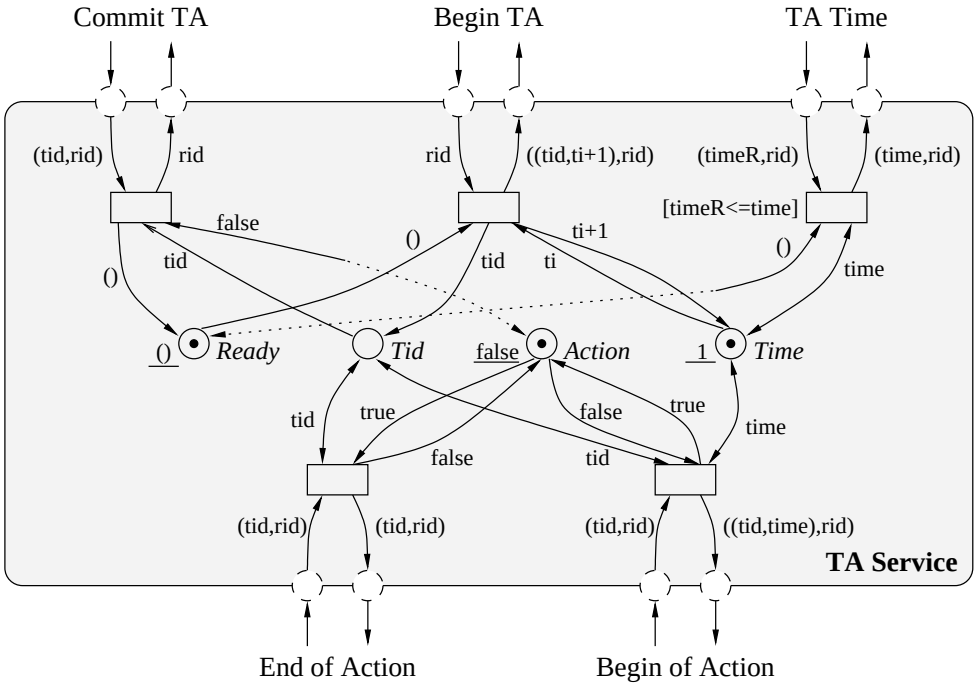


Figure 4.18: Specification of the transaction service component

To reflect the different states of a transaction, we employ four places called 'Ready', 'Tid', 'Action', and 'Time' with the following semantics:

Ready: is of type 'unit' and initially marked by element '()'. It indicates, if a transaction is currently running. If this place is occupied by a token, then no transaction is running, and a new transaction may be started.

Tid: is of type 'int' and initially unmarked. It indicates that a transaction is currently running. An element at this place indicates the transaction identifier. (It might be noted that a 'begin transaction' request generates a random transaction identifier, since variable *tid* is unbounded. However, a counter-based realization would provide the same functionality.)

Action: is of type 'bool' and initially marked by 'false'. If set to 'true', it indicates that a retrieval or update action is currently performed within a transaction. In this case, 'commit transaction' requests are postponed.

Time: is of type 'int' and initially marked by '1'. It indicates the current transaction time. It is incremented at each 'begin transaction' request.

As a standard parameter of each service request, we require a request identification *rid*. It is used to permit concurrent initiations of requests. The request identifier then enables to associate a service response to the requesting client.

Besides transactions, we permit retrieval operations outside transactions. To avoid inconsistencies, retrieval operations are permitted only, if no transaction is currently running. It is realized through a slight extension of the transaction service. For readability, this extension is represented by a separate interaction net in Figure 4.19. Thus, the transaction service should be understood as the union of both interaction nets. The extension provides an opportunity to declare the beginning and end of (retrieval) actions outside transactions. This facility will be used by the 'Retrieve' service of shared data. Thereby, a non-transactional retrieval request simply provides the void element '()' instead of the current transaction identifier.

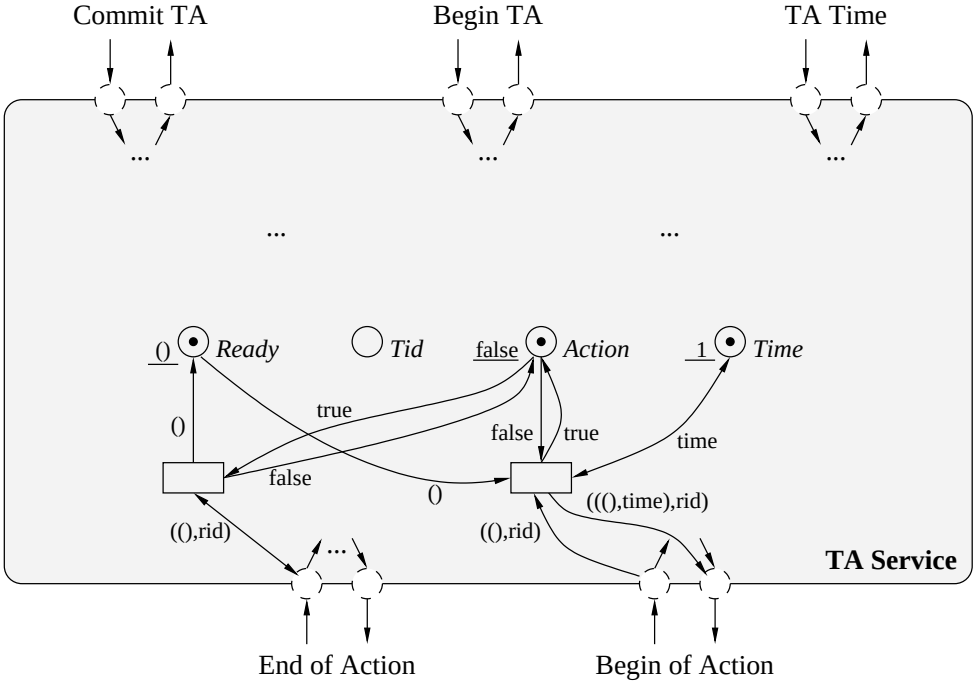


Figure 4.19: Extension of the transaction service component to permit consistent, non-transactional retrieval

Service Request Queues

The services of the ”TA Service” component may be requested by several clients. For example, each rule (at the same level) will apply ’Begin Transaction’ and ’Commit Transaction’ requests. In addition, each ”Shared Data” component (at the same level) will apply ’Begin of Action’ and ’End of Action’ requests. As proposed in Section 4.3.2, we may deploy generic service queues for this purpose. We will deploy two sorts of service queues:

Request Queue: Incoming requests of different clients are arranged within a ”first come, first serve” queue.

Request Merger: All requests are collectively delegated to the service provider. Thus, the service provider may impose an order, if desired.

Both queues serve a different purpose. The ”Request Queue” is deployed, if the service provider equally processes each request. For example, the order of ’Begin Transaction’ requests initiated by different clients is not relevant for the service provider ”TA Service”. The ”Request Merger” is deployed, if the service provider itself needs to impose an order on incoming requests. This situation occurs, if there exist potential requests which cannot be processed immediately be the service provider. However, there might concurrently exist requests which could be processed immediately — but they are waiting in the queue. A ”Request Merger” then prevents such situations of blockage. According to the ”TA Service”, we deploy a ”Request Queue” wrt. ’Begin Transaction’ and a ”Request Merger” wrt. each of the other services.

A specification based on interaction nets is represented in Figures 4.20 and 4.21. It demonstrates the case of two clients which can be generically extended to any number of clients. We represented requests by variable *req* and according responses by variable *res*. Note that besides the number of clients, the data types used for requests and responses are considered as generic parts of the queue specifications. Furthermore, we use request identifiers *rid* to distinguish requests. According to the request merger specified in Figure 4.21, besides identifiers of incoming requests $rid1$ and $rid2$, we generate a new identifier *rid* for each request. It is necessary, since we do not require that request identifiers $rid1$ and $rid2$ of different clients are globally unique. Uniqueness is then provided by identifier *rid*. Such an integration of identifiers is not necessary for the ”Request Queue”, since it exclusively performs one request at a time.

By the specification of their interfaces and dynamic semantics, queues scale up to queues over queues. This property is particularly useful for the delegation of requests over several levels. We employ this opportunity to delegate requests to the transaction service and to persistent shared data over several levels. For example, the transaction service is only realized once at the global level. At local levels, the transaction service represents a rather virtual service, since it only delegates requests to the next higher level. The delegation can be realized by request queues. Since the delegation is applied at each level, we obtain a queue on queue composition as illustrated in Figure 4.22.

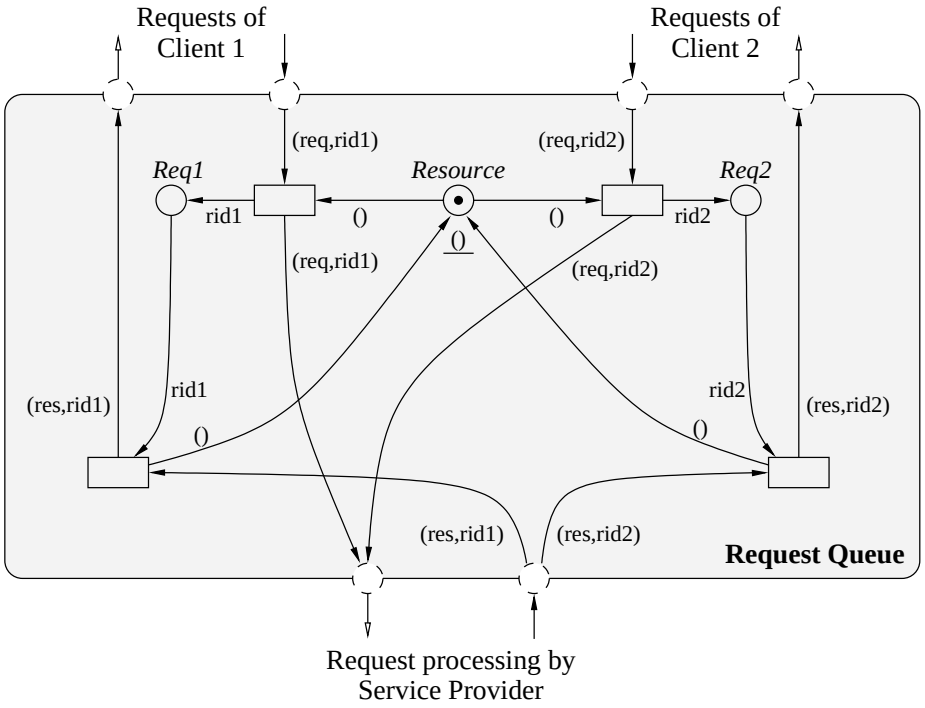


Figure 4.20: Specification of composition component ”Request Queue”

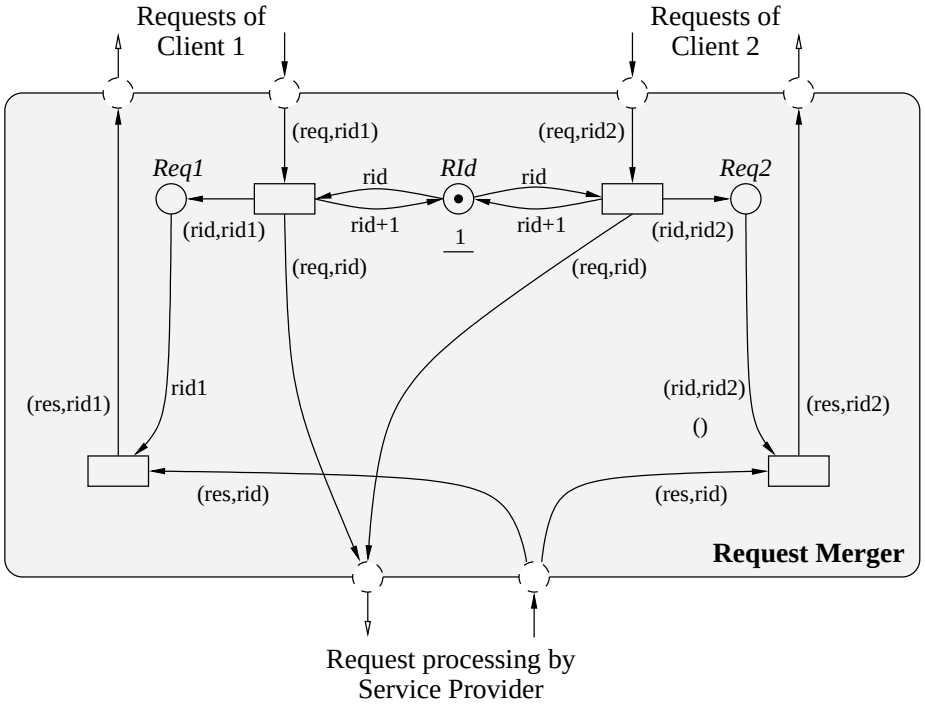


Figure 4.21: Specification of composition component ”Request Merger”

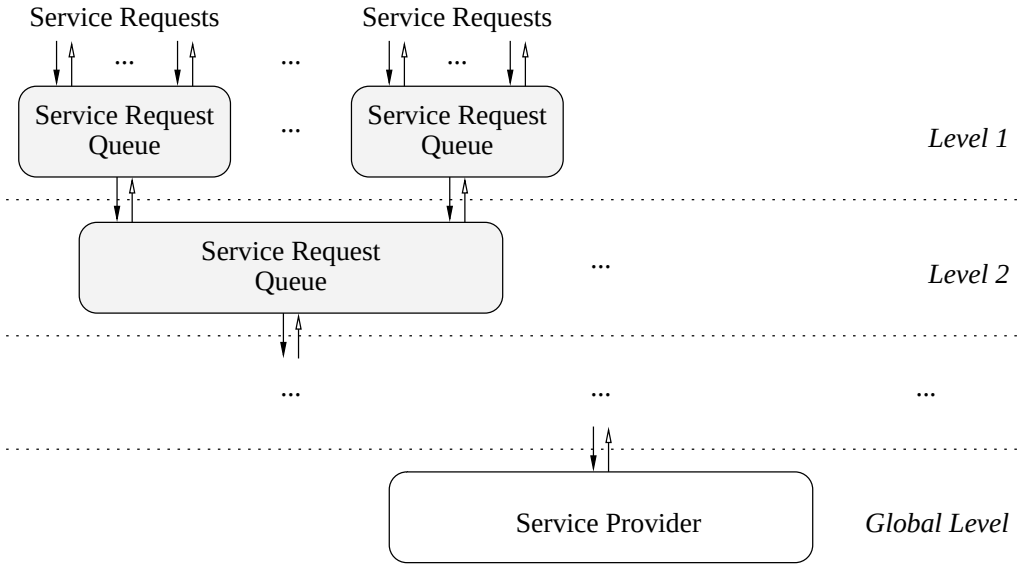


Figure 4.22: Request delegation by use of service request queues

Shared Data

Similarly to the transaction service, (i) shared data declared as public/view is realized by delegation via service request queues, and (ii) shared data declared as private/persistent is realized by an interaction net which ”materializes” according services. Therefore, we only need to specify generic components realizing persistent shared data. These components are generic wrt. the data types used for storing particular data. Any elementary as well as complex data type corresponding to the type system of CP nets may be deployed.

In contrast to the transaction service, services of shared data are not realized autonomously. More precisely, they in turn request services provided by the transaction service to guarantee transaction semantics. As specified in Figure 4.23, a shared data component generally provides three services: (i) ’Update’, (ii) ’Retrieve’, and (iii) ’Updated’. The interaction net employs two places ’Data’ and ’Time of Last Update’. While ’Data’ stores the current value of the shared data, ’Time of Last Update’ stores the transaction time of the last update request. Note that the data type of place ’Data’ represents the generic part of the component.

The services provided by shared data components are realized as follows:

Update: updates the data. It is realized by three transitions. The first transition ’begin’ requests the transaction service to perform an action. Thereby, the transaction identifier is verified and the current transaction time is provided. Afterwards, transition ’update’ performs the actual update of the data and, con-

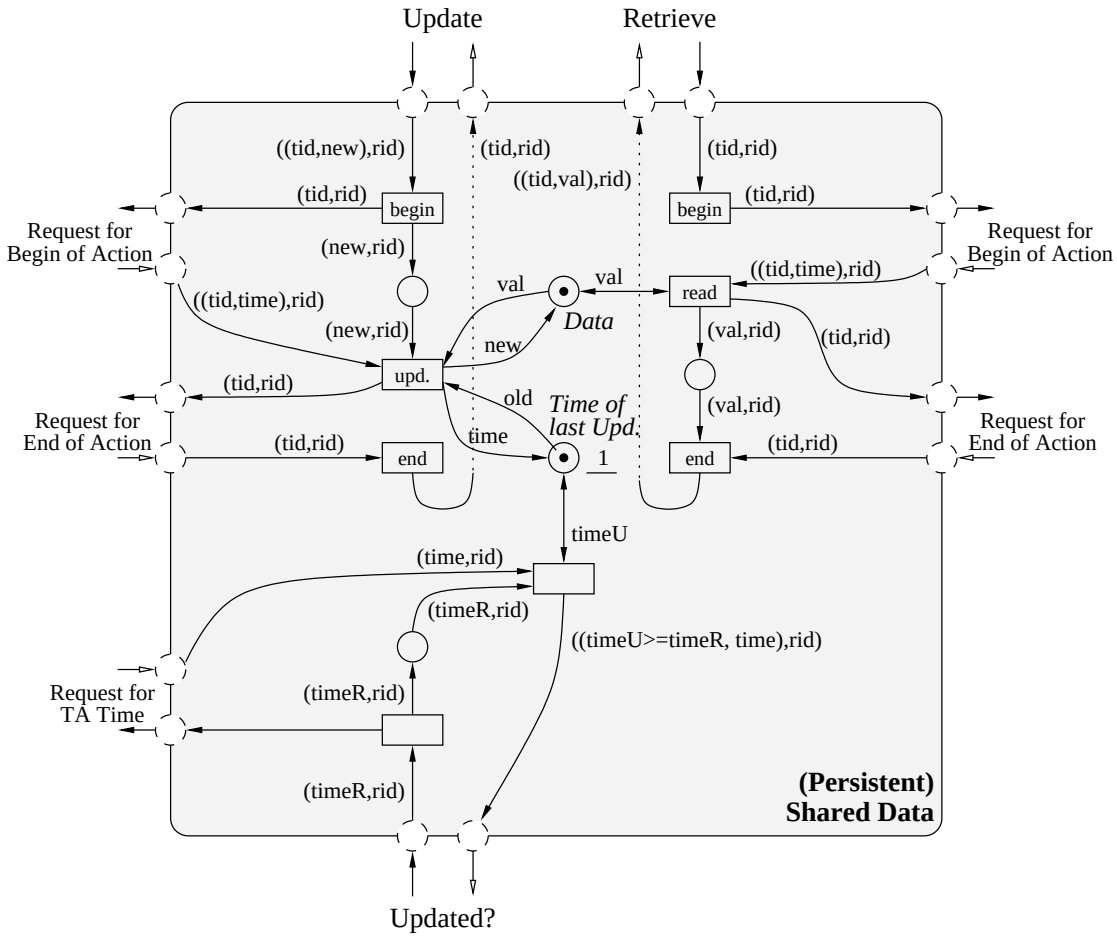


Figure 4.23: Net-based realization of shared data

currently, adjusts 'Time of Last Update'. Finally, transition 'end' informs the transaction service about the accomplishment of the action.

Retrieve: provides the current data. It is realized analogously to service 'Update'. Instead of updating places 'Data' and 'Time of Last Update', it only reads the current data and provides it to the client. In contrast to updates, retrieval is permitted outside transactions as well. In this case, the 'Retrieve' request has to be initiated by the void element '()' instead of the current transaction identifier. Note that a client cannot initiate a non-transactional update request by initiating the void element '()'. It is prohibited by choosing an accordingly restricted data type for the input place of the update interface. It does exclusively permit clients to provide integer values.

Updated: answers the question of whether an update occurred at a given transaction time *timeR* or (timely) afterwards. This service is requested by ECA rules to

realize the 'detection' of events. As will be explained later, each rule stores the transaction time of its last initiation to realize, whether it already processed an event (i.e., an update) or not yet. The 'Updated' service is realized by two transitions. Thereby, the second transition compares the transaction time *timeR* provided by the client with the transaction time of the last update. If the last update was more recent, then 'true' is return. Note that according to the function of service 'Updated', the second transition would suffice already. However, the first transition, prevents from initiating 'Updated' requests without necessity. If 'Updated' requests are initiated through a polling loop, there might occur several requests at the same transaction time. However, if transaction time *timeR* provided by the client is greater than the current transaction time, then the 'TA Time' service will delay a response, until the current transaction time proceeds.

The interaction net presented in Figure 4.23 employs interfaces to services 'Begin of Action' and 'End of Action' twice. The connection to the transaction service component is then realized by deploying service request queues. Although, a net specification with single interfaces is possible as well, this method is more convenient and readable.

4.4.2 Context Transition Model

Both the context transition model of elementary ui-components as well as the context dependency model of ui-composition components are realized by a unified specification. Thus, their distinction is reflected at a semantical level only. As represented in Figure 4.24, the context transition model basically consists of a set of ECA rules — each encapsulated by a single sub-component. The rules access the context (i) to test their activation and (ii) to perform actions, i.e., retrieval and manipulation operations. Besides the rules, there exists a sub-component "Termination Test" which indicates the termination of the rules wrt. a specific level. Note that this service only indicates the current status of the rule system — being either "running" or "terminated". It does not verify if a rule system will in fact terminate. Thereby, termination is identified wrt. a level, if no rule at this level is activated and termination wrt. the next lower level is indicated. Consequently, if termination is indicated by the global level, then no rule of any ui-component is activated.

Rules

ECA rules are specified by generic interaction nets. One sub-net realizes the recognition of events and another sub-net specifies the condition and action part by initiating an according transaction. To realize the termination test, each ECA rule maintains a transaction time t_{stable} ('last stable time') which is continuously updated. This time indicates the transaction time at which two conditions were fulfilled at last:

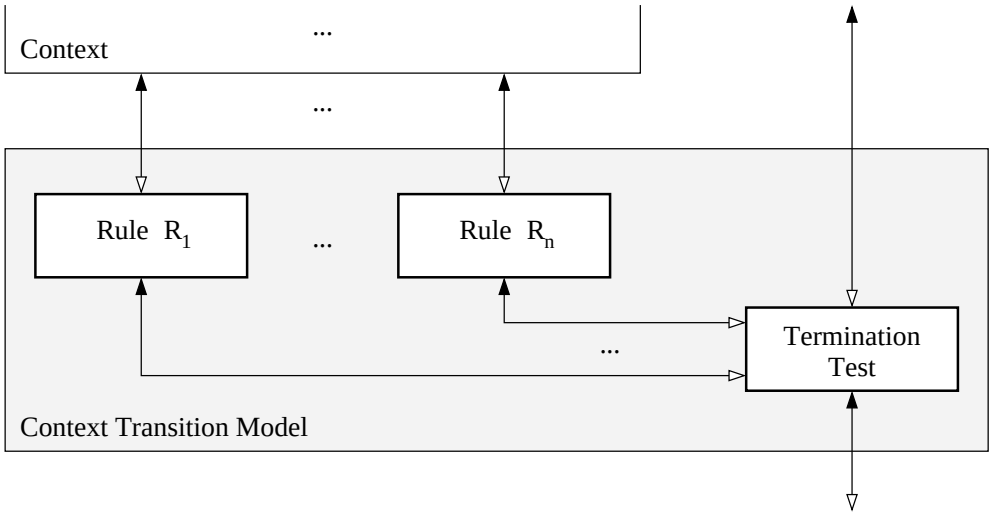


Figure 4.24: Generic structure of the context transition model

(a) the rule was testing an occurrence of an event at time t_{stable}, and

(b) the test was negative, i.e., the rule was not activated at time t_{stable}.

This time can be requested by a client through a 'TA Time' service. Analogously to the 'TA Time' services of the transaction service and shared data, it is realized as a call back service. Therefore, a client has to provide a transaction time t_{req}, when it requests the 'TA Time' service. If t_{req} is greater then transaction time t_{stable}, then the client will be informed only after time t_{stable} advances. It will be at the earliest at time t_{req}. However, it might be later, if the rule becomes activated at time t_{req}. In this case, time t_{stable} is not advanced immediately, but the rule body is executed by initiating an according transaction.

In turn, each "Termination Test" component provides an according service called 'Terminated'. It waits for responses of 'TA Time' services of (i) all rules of this level and (ii) all transaction tests of the next lower level. If all responded transaction times are equal, then this time is reported. Otherwise, the request is restarted with the maximum time t_{max} of all received transaction times. Note that if a request returns a time smaller than t_{max}, it indicates that an according rule did not perform an activation test at transaction time t_{max}. Therefore, the test must be restarted for time t_{max}.

If a 'TA Time' request of the transaction test at the global level is finally responded by an transaction time t_{fin}, then all rules terminated at this time. The received final transaction time t_{fin} states that each rule at each level performed an activation test at time t_{fin} which were reported by a negative result. Thus, at time t_{fin} no rule is activated. In contradiction, we might assume that there exists a rule R_1 which is activated at time

t_{fin}. However, in this case R_1 would not report time t_{fin} as described above through conditions (a) and (b).

According to a net-based specification, an ECA rule

```
ON updated(<X>)
IF <condition>
DO <action>
```

is generically realized by polling the following loop. Note that the loop is continuously executed independent from client requests initiated through the 'TA Time' interface.

1. Send an 'Updated' request to shared data '<X>'. It returns a truth value and the current transaction time t_{test}. The truth value states, whether '<X>' was updated after the rule was sending the last 'Updated' request.

2. If it returns 'false' (meaning that there did not occur an intermediate update), then advance the 'stable time' by assigning: $t_{stable} := t_{test}$ and restart the loop at 1.

   Otherwise continue.

3. Send a 'Begin Transaction' request to the "TA Service" component.

4. Execute an according sub-net which realizes the condition and action part of the ECA rule. It represents the generic part of the rule. Retrieval and manipulation requests specified by the body of the rule are realized through according interfaces to the context. Their specified control flow is accordingly reflected by the sub-net. Note that CP nets are Turing complete, such that necessary control flow constructs can be transformed into an interaction net specification.

5. Send a 'Commit Transaction' request to the "TA Service" component and restart the loop at 1. Thereby, the 'stable time' is not advanced. It will be updated only, after the rule reached a stable state, i.e., after a subsequent 'Updated' request returns 'false'.

An interaction net specification realizing an ECA rule is represented in Figure 4.25. Service 'TA Time' provides a call back service. Given a transaction time *timeR*, it informs the client as soon as the 'stable time' (which is continuously updated by the loop) reaches or exceeds *timeR*. Thereby, the loop starts at transition 'event' and finishes at transition 'end'. Transition 'event' ensures that the test, whether an associated event did occur, is performed at most once at each transaction time. According to internally initiated requests, we randomly chose request identifier 1. It is not significant, since we initiate requests sequentially.

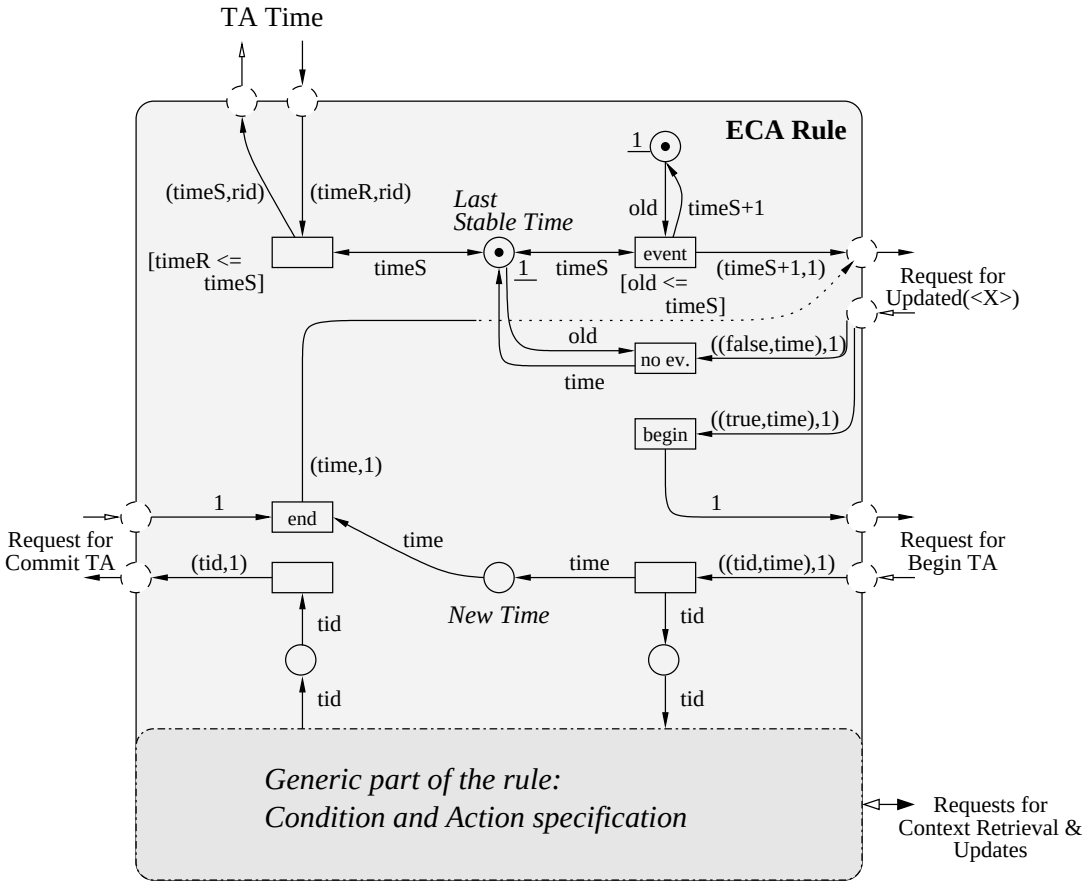


Figure 4.25: Net-based realization of ECA rules

The interaction net is generic at two respects. Firstly, an 'Updated(<X>)' request is initiated to the particular context specified by the event clause of a rule. Secondly, condition and action clause of a rule are specified by a sub-net. Thereby, the sub-net (i) initiates according context retrieval and manipulation requests and (ii) determines the control flow. Note that an according sub-net may be generated from an appropriate rule specification language resembling the pseudo-code syntax deployed in Section 4.2.

Termination Test

Analogously to the "TA Time" service provided by ECA rules, the "Termination Test" component maintains a transaction time t_{stable} ('last stable time') which is continuously updated. It indicates the transaction time at which (i) all rules at this level and (ii) all rules of the next lower level reached a stable state. This time can be requested by a client through an 'TA Time' service. It is again realized as a call back service.

Therefore, a client has to provide a transaction time t_{req}, when it requests the service. It will be informed as soon as the next stable state is reached.

Figure 4.26, represents an interaction net specification for the case of two rules and two lower-level ui-components. Thereby, a polling loop (starting at transition 'req') continuously updates time t_{stable}, if the next stable state is reached. The loop initiates requests (i) to the 'TA Time' service of all rules at this level and (ii) to the 'TA Time' service of all "Termination Test" components of the next lower level. For each request, a transaction time of the next stable state is received. If all responded transaction times are equal, then a new stable state is reached and time t_{stable} is advanced (by transition 'upd'). Otherwise, the loop is restarted with the maximal transaction time received.

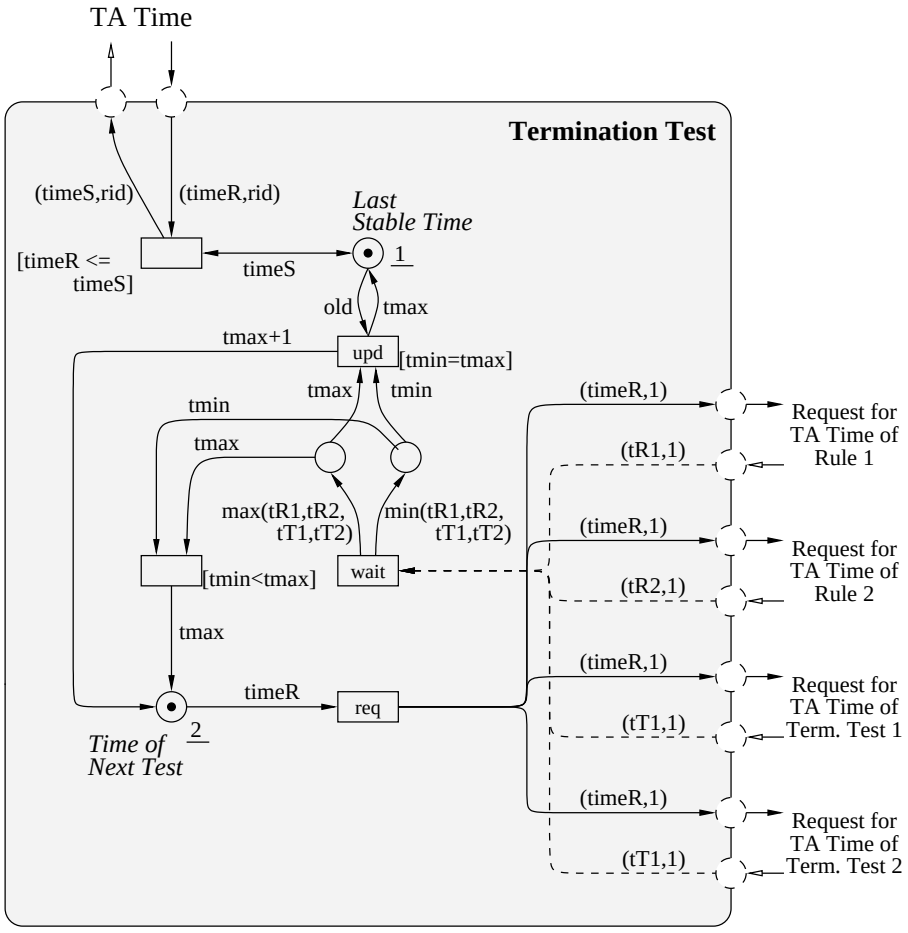


Figure 4.26: Net-based realization of the "Termination Test"

After a user event was initiated, the 'TA Time' service at the global level reports about the termination of the complete rule system. Since "UI-View" (sub-)components of each ui-component require this termination information to provide consistent views, the 'TA Time' service of the global "Termination Test" will be accessible by all ui-

components. To avoid ambiguity, we call this global 'TA Time' service *'Terminated'*. It will be provided by each local "Termination Test" component through delegation (as already applied for shared data as well as the transaction service).

4.4.3 UI-View

An "UI-View" component provides an interface 'UI-Spec'. As soon as the rule system terminates, a dialog specification is generated wrt. the current context. This specification is then provided to the next higher level. There, it is consumed either (i) by the "UI-View Composition" component of the next higher level, or (ii) by the "UI-Controller" at the global level. Figure 4.27 represents an according interaction net specification. By requesting the 'Terminated' service, it delays processing as long as transition rules are adapting the context according to an initiated user event. Afterwards, a corresponding dialog specification is generated by a sub-net. This sub-net represents the generic part of the "UI-View" component. To provide the current dialog specification, it initiates non-transactional retrieval requests to the contexts required. Note that consistency is still guaranteed, since non-transactional retrieval requests are delayed as long as a transaction is running. However, at the time the dialog specification is computed, there are running transactions, since all rules terminated.

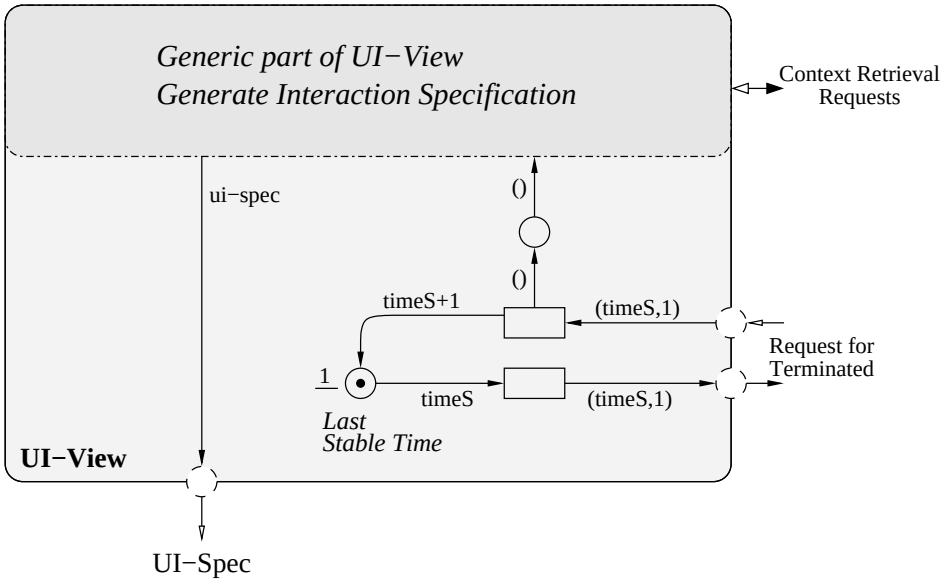


Figure 4.27: Net-based realization of the "UI-View" component

Concrete syntax and semantics of generated interaction specifications depends on the controller deployed. We will employ the following unified representation adapted from [Lew00, VLF00]. Commonly, there exist different input and output types which may

be used. Typical input types comprise, for example, character strings, integers, real numbers, complex forms, or choices. Exemplary output types are headline, table, character string, number, picture, audio, or video. They are usually further specified by parameters. For example, an input type 'choice' expects the actual choice elements as a parameter. The following elementary structure comprises these opportunities:

Output specification: List of (Output identifier, Output type, Parameters)

Input specification: List of (Input identifier, Input type, Parameters)

Style information: List of (Style identifier, Style type, Parameters)

Identifiers may be used for referencing. For example, style information of type 'Input/Output Association' might associate input elements with output elements by referring to their identifiers. According to the type system of CP nets, we may apply a relational style for representing interaction specifications. Thereby, specifications are represented by data type 'ui-spec' defined as 'ui-spec := tuple(list(entry), list(entry), list(entry))', assumed that 'entry := tuple(string, string, list(list(string)))'. Data type 'entry' then corresponds to a relational representation of parameters. Note that type 'ui-spec' is sufficiently expressive to represent any dialog specification of the style proposed above. To give an example, a concrete dialog specification generated by ui-component 'List Scrolling' reads as follows:

```
(
  ('Out:View',  'STRINGLIST', [['Faust', 'Carmina Burana', 'Latin Night']]),
  ('In:Scroll', 'CHOICE',     [['first', 'last', 'next', 'prev']]),
  ()
)
```

where style information is neglected. Thereby, 'STRINGLIST' and 'CHOICE' represent output and input types respectively which must be understood by the controller. Note that interaction specifications are not restricted to user interaction. They may specify arbitrary interaction as, for example, interaction with devices as printers as well as interaction with remote clients. However, throughout the thesis, we primarily focus on aspects of user interaction only.

Sub-component "UI-View Composition" of ui-composition components roughly corresponds to the realization of "UI-View". As a slight extension, it possesses an interface to ui-components of the next lower level. Therewith, it receives according dialog specifications. Together with possible context retrieval requests a composite interaction specification is generated.

4.4.4 Assertions

By the introduced specification, ui-components realize a step-wise processing of elementary user activities. UI-components (i) receive events initiated by the user, (ii) process them by means of context adaptation, and (iii) provide output to the user and activate associated events.

Receiving user events: a ui-component initiates external requests according to the events it has been activating, and waits for responses from the environment.

Adapting dialog context: At occurrence of an event, an associated context transition rule initiates a context change. Iteratively other context transition rules may be initiated in turn.

Providing user output and activating events: After context adaptation has been finished (recognized by the termination of the context rules), an according dialog specification is generated and provided to the environment.

At the global level, interaction between ui-components and their environment is limited to the request of initiated user events and a subsequent generation of the next dialog specification. Necessary context adaptations are realized internally. To guarantee that ui-components function as intended, the following assertions must be imposed onto their environment:

- Requests for events have to be answered completely. More precisely, an event initiation is indicated by the environment (i) by positively answering the request associated with the initiated event, and (ii) by negatively answering all other event requests. If an environment deploys a shared data realization introduced in Section 4.4.1, this assertion will be satisfied.

- A subsequent event initiation may only be indicated by the environment, after it received the next dialog specification from the ui-component.

At a component perspective, we therefore do not necessarily require a concrete specification of the environment of ui-components. If an environment is provided, we exclusively need to verify, if this environment satisfies above assertions. In this case, the assertions have to be formalized in terms of properties (cf. 3.3.2). Subsequently, properties of ui-components can be derived based on environmental assertions.

Although, an environment can be characterized in terms of dynamic requirements, we discuss a concrete environment realization in the following. Thereby, it will provide desirable opportunities for simulation and prototyping.

4.4.5 Simulation and Prototyping

In this section, we propose two distinguished alternatives for the realization of environments. The responsibility of the environment basically comprises the interpretation of the dialog specifications, receiving user events (corresponding to the specification), and providing events to ui-components.

Driver-based Realization

The first alternative is based on driver implementations adopted from [CL99, Lew00, Hei01]. It is mainly intended to provide an opportunity for rapid prototyping. Thereby, a user may test (complex and adapted) ui-components wrt. a particular user interface. It is supported by drivers which represent concrete implementations wrt. particular programming languages and user interfaces. As illustrated in Figure 4.28, a driver receives as input a (declarative) dialog specification, transforms it into directives of a respective user interface, and initiates these directives to the user interface. After an event is initiated by the user, the driver receives the event and returns it to the controller. Thereby, the controller degenerates to a rudimentary interaction net which exclusively delegates the received dialog specification to the driver and publishes the user event received from the driver to the ui-component.

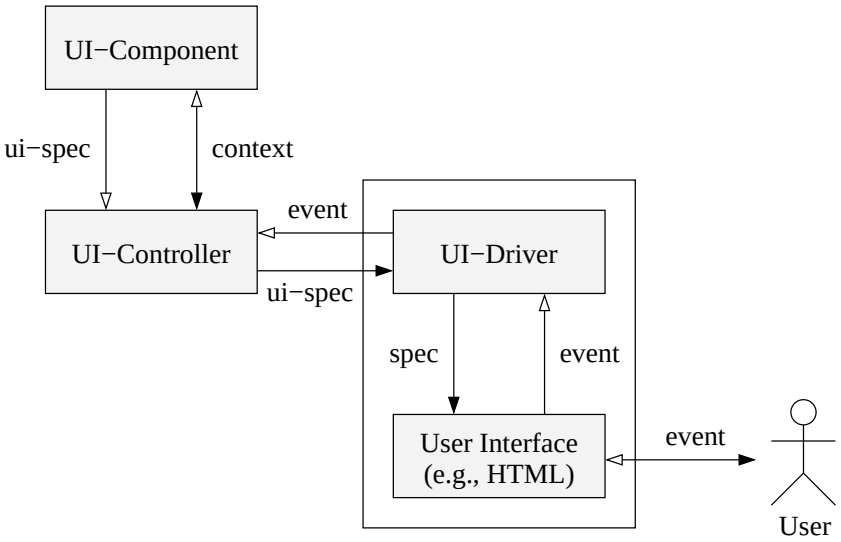


Figure 4.28: Driver-based realization of user interaction

Consider for example a driver which supports Web interfaces. The specification it receives from an interaction net is transformed into a HTML page. Thereby, associated style information is regarded. Afterwards this page is transmitted to a web browser. If a user initiates an according event, for example, by clicking on a link or submitting

a form, the driver is reactivated and provides the according event to the requesting interaction net. Note that an according facility of interaction between drivers (representing executable code) and interaction nets is supported. For example, the CP net simulator provided by [Jen02] permits so-called code segments. They can be utilized to initiate associated drivers. (As a remark, code segments are not intended to generalize CP nets. They were introduced rather to extend the flexibility of net simulation.) At the codesign project [CT97, Lew00, VLF00], the concept of user interface drivers which are initiated by net-based specifications has been introduced and prototypically implemented for command line interfaces, TCL/TK interfaces, and Web interfaces [See98, Rad01, Hei01].

Net-Based Realization

A realization of the environment based on interaction nets intends to support simulation as well as verification of ui-components. As illustrated in Section 4.1, it is divided into a user interface controller and a user component. We simulate user interaction within two consecutive steps:

Event Selection: A (possibly adapted) dialog specification is provided to the user component. An actual user or a predefined simulation scenario may choose an event. It is realized by selecting one of the event identifiers occurring at the specification. The selected event identifier is then returned to the controller.

Event Specification: After an event is chosen by the user, it must be specified further, if required. For example, if a character string or a choice list was selected, the actual string or choice must be provided by the user. It can analogously be provided by an actual user or predefined simulation scenario. The choice is then returned to the controller.

Although a two step approach might look a bit circumstantial, there exists a meaningful correspondence to practical user interaction. The first step of selecting a user event corresponds, for example, to moving the mouse pointer or keyboard cursor to the position of a string field, input form, or link item. The second step then corresponds to entering a string, filling a form, or clicking a link.

Accordingly, we divide a user component into two (sub-)components: event selection and event specification (cf. Figure 4.29). While event selection corresponds to a choice wrt. a finite set of elements, event specification depends on the input type selected. As indicated by the following exemplary list of selected input types, they possesses distinguished semantics.

'VOID' is represented by data type 'unit'. It is used to provide unspecified events. At concrete user interfaces they correspond to links or buttons.

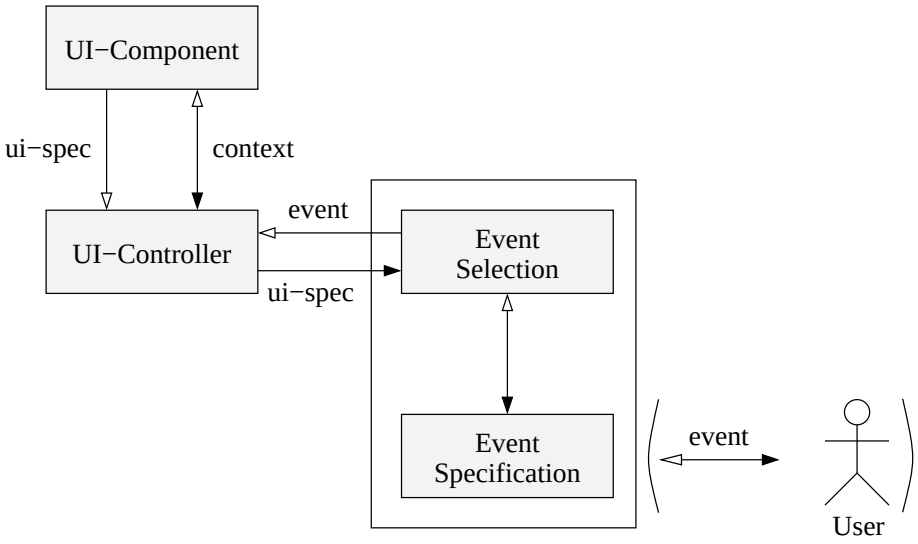


Figure 4.29: Net-based realization of user interaction

'BOOL' is represented by data type 'bool'. It commonly expresses yes/no decisions.

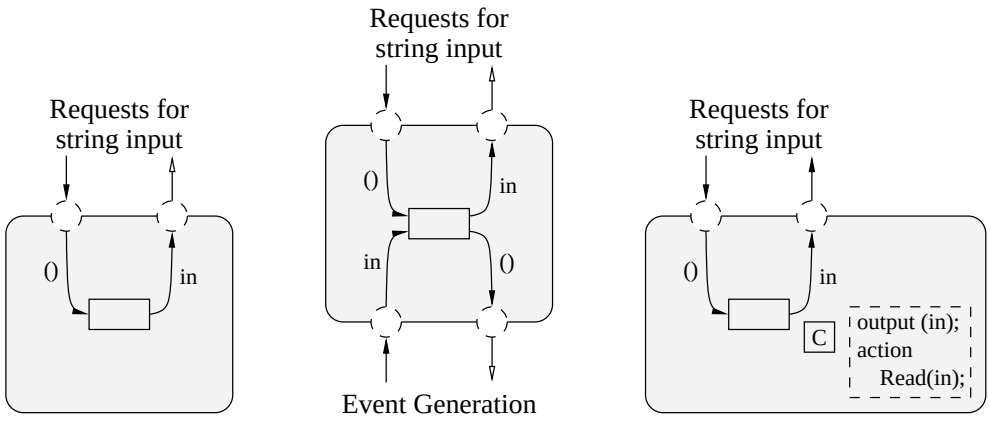
'STRING' is represented by data type 'string'.

'CHOICE' is represented by data type 'subset(int)'. It represents a single selection from a finite list of elements. From the perspective of event specification, it corresponds to selecting an integer from a finite set of integers. Thereby, the integer represents the position in the list or an unique element identifier.

'N-CHOICE' is represented by data type 'list(subset(int))'. It represents a choice of a sublist from a finite list of elements. For example, choosing a list of desired departure airports corresponds to this type.

To support their differences, a sub-net may be provided for each input type. Thereby, these sub-net specify the constraints imposed by the associated input type. Figure 4.30 exemplarily illustrates a realization wrt. input type 'string'. Depending on the mode of usage, we represent three realizations: (i) simulation of random scenarios, (ii) simulation of predefined scenarios, and (iii) simulation with concrete user interaction. Figure 4.30(a) permits the execution of random scenarios. We assume that the output place of the interaction net is of data type string. Since variable *in* is unbounded, an arbitrary string will be provided on request. Since CP net simulators do not commonly cope with unbounded variables, an alternative realization based on code segments may be chosen for simulation purposes. Figure 4.30(b) permits the execution of predefined scenarios. Thereby, an associated simulator net generates events according to its configuration. Thus, on request the next event of the predefined scenario is provided. Finally , Figure

4.30(c) permits concrete user interaction. Thereby, a user may initiate events during net simulation. It is realized by an external function call 'Read()' which is executed through a defined code segments. Function 'Read()' initiates a request to the user. Subsequently, user's input is associated with variable *in* and provided as output of the interaction net.



(a) String input: Random scenarios

(b) String input: Predefined scenarios

(c) String input: Interactive scenarios

Figure 4.30: Specification of input type 'string'

Note that the restriction imposed by input type 'string', is specified by associating variable *in* as well as the according output places by data type 'string'. However, arbitrary complex restrictions may be specified by complex sub-nets.

4.5 Verification of Ergonomic Aspects

In this section, we investigate how far quality of user interaction can be verified by analyzing interaction nets. We do not concern analysis of dynamic net properties in its general sense as this is an independent research topic (cf. [GV02, Bau96, Jen97a]). We rather focus on the question:

> *"Which particular aspects of user interaction can be verified, if interaction specification is based on the introduced formalism of ui-components?"*

In particular, we investigate which and how far qualitative aspects of *interface independent* specifications can be verified. Therefore, we neglect interface dependent

properties as, for example, arrangement of controls and displays. Associated issues and approaches can be found in text books as, for example, [DFAB98, Nie94, Nie00].

Before we consider specific quality aspects in detail, we discuss general requirements for their verification. According to task representation, we adopt the model of hierarchical task analysis (HTA) introduced by Duncan et. al. [AD67]. Thereby, complex user tasks may consist of several sub-tasks. Depending on the desired level of detail, this task division may be applied successively which results in a task hierarchy, i.e., a hierarchy of sub-tasks required to accomplish a task. In addition, there might be alternative opportunities to accomplish a single task. To verify, if a task specification coincides with an interaction specification, a relation between the task hierarchy and the interaction specification must be established. For example, the web-site maintenance system ”TANGOW” [CPR01] applies a specific relation between tasks and specified dialog structures. There, elementary tasks (called simple tasks) are associated with single pages. More precisely, they define a hierarchic web-site structure and associate simple tasks with leaf pages and composite tasks with non-leaf pages.

Generally, there exist several alternatives to associate sub-tasks with interaction specifications. Commonly, elementary tasks are associated with the notion of dialog step (or activity). Thereby, a dialog step can be characterized by

(i) the output provided by the system together with events initiated by the user, or

(ii) changes of the system state.

While (i) characterizes dialog steps at an external perspective of the user, the second associates dialog steps with an internal perspective. Reconsider the example above. Their established relation between tasks and interaction specifications corresponds to characterization (i). Thereby, Web pages correspond to elementary dialog steps. According HTML code comprises the specification of system output and enabled user events. They associate complex tasks with inner nodes which corresponds to (sub-)trees. Since these sub-trees may be browsed in different ways, complex tasks correspond to (alternative) sequences of dialog steps.

Depending on the chosen perspective, dialog steps are reflected by the specification of ui-components as follows.

External perspective: Dialog steps at the external perspective correspond to elementary dialog specifications generated by ui-components. They comprise system output as well as enabled user events according to a particular dialog situation. To be precise, a dialog step consists of a dialog specification together with an event initiated by the user. This pair reflects the dynamic characteristics of dialog steps. However, if confusion is excluded, we commonly simply identify dialog steps with elementary dialog specifications.

Internal perspective: A stable system state of ui-components is characterized by the state of the context after the termination of the rule system. More precisely, the markings of the 'Data' places of all context components identify the current system state. Therefore, changes of the system state correspond to changes of the context between two consecutive stable states (wrt. the rule system).

In the following, we primarily focus on the external perspective. Thus, we may simply refer to elementary dialog specifications as dialog steps. We will analyze specific qualitative aspects of user interaction which permit verification based on interface independent specifications. Thereby, considered aspects and factors are not intended to be complete. We rather focus on selected aspects of quality where we identified possible approaches according to verification.

Ergonomics

Dix et. al. [DFAB98] characterize the issue of ergonomics as:

> *"A primary focus [of ergonomics] is on user performance and how the interface enhances or detracts from this.*"

Ergonomics is affected by the following aspects:

Efficiency

Efficiency of user interaction concerns the time users need to accomplish certain tasks. It is an essential quality criteria of interfaces, in particular, if tasks are performed very often. The following factors primarily affect efficiency.

> (Q1) "How many dialog steps does a user need to perform to accomplish a particular task?"

Verification approach: The condition of task accomplishment can be stated in different terms: (a) in terms of sequences of sub-tasks, (b) in terms of final sub-tasks, or (c) in terms of final system states. In case (a), we need to investigate user scenarios by means of sequences of initiated user events. Thereby, each scenario corresponds to a particular sequence of dialog steps. If an associated sequence of dialog steps corresponds to an appropriate sub-task sequence that performs the task, its length indicates the requested number. Since dialog steps are externally perceivable, the external (black-box) behavior of ui-components is sufficient for determining this number. In particular, if according behavioral properties are determined, the possible scenarios can be derived without analyzing the net specification. It is particularly beneficial in

the case of composite ui-components. In this case, properties of ui-components can iteratively be derived from properties of according sub-components.

Case (b) is verified analogously to case (a). In contrast, permitted sequences of dialog steps do not need to be verified against particular sub-task sequences. It is sufficient to verify that the final sub-task is accomplished.

In case (c), task accomplishment is stated in terms of a context state (or set of context states) which corresponds to the dialog situation of task completion. The approach of determining according scenarios resembles that of case (a). In contrast, the recognition if a task is accomplished, is realized by analyzing intermediate context states. Since context changes are commonly performed internally, according interaction net specifications must be analyzed.

Besides the number of dialog steps, their particular durations affect efficiency of task accomplishment.

(Q2) "What time does a user need to perform single dialog steps?"

Verification approach: Generally, semantic information about the interactive complexity of dialog steps is required. There is no automatic way to deduce the time needed to perform dialog steps. At selected specifications, it might be reasonable to deduce the time from the complexity of the input types. However, answering a boolean typed "yes/no" question might take the same effort as answering questions of complex input types. In addition, since we permit abstract user events, elementary input types may correspond to complex user activities.

To determine a time estimate for an interactive task, the designer must define approximate duration for each dialog step. This task can be relieved by introducing certain classes as, for example, "simple fact question" (as requests for name or birthday), "simple choices" (as requesting a country out of a list), or "decisions" (as requesting to decide the seating in a musical play based on price and assumed impression). Classes are associated by certain times and can be assigned to dialog steps. The total time of a scenario then consists of the cumulative duration of the interactive part of dialog steps (assumed the duration for performing initiated system actions is comparably short). Applying this procedure, verification can be reduced to the approach proposed for (Q1).

Mental workload

Mental workload concerns the amount of information a user needs to keep in mind while performing an interactive task [HLP97, DFAB98]. For example, while answering an email request, the user of an email system must keep in mind the content of

the email read before. To decrease mental workload, dialog steps are commonly augmented by context information (or escort information, see [FST98]). Context information is employed to recapitulate and summarize information from the dialog history which is required to perform the current dialog step. For example, email systems commonly provide a visual copy of emails while their response is being prepared.

(Q3) "Does mental workload exceed a critical threshold?"

Verification approach: Mental workload that is imposed onto a user to perform a dialog step within a particular dialog situation can roughly be determined by the difference between the infons required to perform the dialog step and the infons provided at this step as output to the user. The intuition behind is that information conceivable directly reduces mental workload, while information that must be kept in user's short-term memory temporarily increases it. However, providing every possible information at each dialog step reduces comprehensibility as well.

As an approach, the designer associates dialog steps by infons required to perform this step. At the verification, this information can be compared with the corresponding user output. The inherent problem is that needed information is usually expressed in abstract terms as, for example, "seating information for a musical play". In contrast, user output is given at runtime in concrete terms as, for example, "(Les Miserables, Dress circle, 70 Euro)". Therefore, an appropriate representation of infons is required that allows to derive according associations. As an opportunity, according semantic information may be associated by means of style information.

Completeness

Information systems are designed to solve specific interactive tasks. If all intended tasks can be interactively performed, the information system is complete in that respect.

(Q4) "Does there exist user scenarios for accomplishing all required tasks?"

Verification approach: The verification approach proposed for (Q1) applies for this question. Thereby, given a task, the approach determines corresponding user scenarios. To positively answer question (Q4), the existence of user scenarios must be verified for each task.

(Q5) "Does an interaction specification completely realize a task model?"

Verification approach: In contrast to question (Q4), this question intends to verify, if there exist user scenarios for *each* particular sequence of sub-tasks specified at the task

model. Figure 4.31 illustrates a counter-example. Assumed A, B, C represent sub-tasks, and the task model specifies the accomplishment of a task T by *any* sequence of of length 3 that contains sub-tasks A, B, and C. Figure 4.31 proposes a dialog specification where nodes represent system states (or dialog situations) and arcs represent dialog step (corresponding to the sub-tasks). Besides others the dialog specification does not support sequence $B \rightarrow C \rightarrow A$. In other words, if a user performs sub-task B and reaches dialog situation s_B, (s)he will fail in continuing by sub-task C, since C is not provided in s_B. Obviously, such situations decrease efficiency and may even prevent users from accomplishing their intended tasks at all.

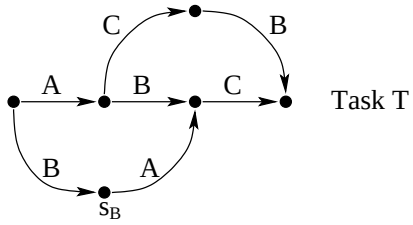


Figure 4.31: An exemplary dialog structure with missing continuations

Its verification reduces to that of (Q1). Thereby, for each sub-task sequence of the task model, it must be verified, if there exists a (finite) user scenario.

Consistency

In [DFAB98], Dix et. al. state:

> *"Consistency is probably the most widely mentioned principle in the literature on user interface design. [...] Consistency is not a single property of an interactive system that is either satisfied or not satisfied. Instead, consistency must be applied* relative *to something."*

To enable meaningful interaction, system responses should closely relate to user's intentions (or expectations) at each dialog situation.

(Q6) "Do database views correspond to user's mental model?"

Dynamic system responses are usually provided through database views. These views must correspond to the user's mental model at each dialog situation. More precisely, contextual restrictions agreed upon in the course of a dialog must be obeyed during response generation. For example, a travel planning service which intends to provide the user with a list of events should primarily focus on events that take place during

the time period of the scheduled journey and which are closely located to the travel destination. Thereby, the system regards the context of time and location which were provided by the user at a first step of the dialog.

Verification approach: Because of its inherent complexity, a verification of the question cannot be provided for the general case. However, the problem may partially be solved by selected restrictions. For example, at a simplified mental model of the user, sub-tasks are associated by contexts. Similarly, [McC93] propose an abstraction according to the representation of user scenarios of ”query and reply”. They are specified by sequences of pairs of an utterance and an associated *discourse context*. Thereby, the discourse context comprises statements which are assumed to be true at a certain dialog situation. As an elementary opportunity, we identify discourse contexts by plain (context) names. At the example stated above, a context name ’Location’ will be introduced into the discourse context, after it is given by a user. Thereby, the association of a context name with a sub-task is interpreted as: the scope of a user is restricted by this context. We then consider database views as consistent with user’s context, if they are restricted by this context. According to ui-components, restrictions on database views are specified in terms of context dependency rules. To verify view consistency, according rules must be analyzed, if they correctly specify the adaptation of database views based on user’s context. Consider, for example, ui-component ”Single-Level Catalog” introduced in Section 4.3.1. There, the element list *NavList* presented to the user represents a database view. The selected category *SelCat* might be identified as user’s context. To verify, if the view *NavList* correctly adapts to context *SelCat*, according context rules which specify a dependency between *SelCat* and *NavList* must be evaluated.

(Q7) ”Is the interaction specification consistent wrt. the task model?”

This question represents the counterpart of question (Q5). While (Q5) verifies completeness of interaction specifications, (Q7) verifies its consistency. Thereby, consistency wrt. the task model is violated, if there exists a user scenario whose corresponding sub-task sequence is not specified by the task model.

Verification approach: Its verification reduces to that of (Q1). In contrast, the *complementary task model* is analyzed. Thereby, the complementary task model comprises all task sequences which are not permitted by the task model. A positive answer to the question is provided, if there exists no user scenario wrt. the complementary task model.

Security

Although security does not only relate to user interaction, we provide a brief discussion. It is commonly agreed that security can be improved by combining different

methods. While many aspects are related with different areas, one is concerned with user interaction. An essential aspect of security is to provide a view mechanism to prevent users from accessing data and/or processes in an unrestricted way. It is commonly realized by associating users with roles. For each role, specific system rights are defined which determine how far certain (sub-)tasks may be performed by a corresponding user.

(Q8) "Does the interaction specification coincide with security-related restrictions?"

Validation approach: It can be reduced to the problem of investigating which sequences of sub-tasks are available for which roles. For each role, according restricted task models must be verified wrt. consistency. It ensures that sequences of sub-tasks which are not explicitly specified at the task model are not executable. Therefore, the problem reduces to question (Q7).

Sometimes particular sub-tasks shall exhibit distinguished behavior depending on the role of the current user. A typical example are views. A view associated with a sub-task reflects different restrictions depending on user's role. This problem can be reduced to the view consistency problem posed in question (Q6). Thereby, we introduce the current role as a particular context. It then must be verified, if views correctly adapt to context 'role'.

4.6 A Case Study: Interactive Catalogs

In this section, we discuss a possible approach to develop a complex ui-component which realizes an interaction pattern called "interactive catalog". We stipulate the following delimitation of "Interactive Catalogs".

Interaction pattern: [Interactive Catalog]

Problem statement

An interactive catalog provides a user with opportunities to interactively search a collection for desired entries.

Examples

[amazon.com]: provides products. There concurrently exist several opportunities for adaptation of scope: (i) selecting specific product categories as, for example, books, (ii) searching by forms, (iii) searching by keywords, or (iv) following context dependent suggestions.

[bahn.de]: provides train connections. A rough opportunity of refinement and enhancement of scope is realized by a form based interface. A more fine-grained opportunity of adapting scope is subsequently provided by 'enhance' and 'select' operations on the list of suggested train connections.

[cottbus.de]: provides events with the following interactive opportunities: (i) following predefined-defined views as, for example, events of today or this week, (ii) searching by forms, (iii) scrolling forward/backward.

[zdnet.com]: provides daily news. Search is supported by (i) selecting a specific day from a calendar, (ii) navigating to news of the preceding/following day, (iii) selecting a topic of interest, or (iv) providing a keyword.

Solution

Employ the concept of a *virtual view* over the collection. This view is virtual in the sense that it is not necessarily presented to the user permanently. At the beginning of interaction, the view comprises the entire collection. The user may successively adapt the view to his/her needs by applying operations on it. These operations essentially comprise refinement and enhancement of the view. Popular opportunities to realize refinement are based on categories, forms, keywords, and scrolling. Depending on the state of interaction, the view may be hidden, presented partially, or presented completely.

Related patterns

Generally, view refinement and enhancement can be based on patterns [Selectable Search Space] and [Interaction History] respectively. Pattern [Set-Based Navigation] provides a more fine-grained opportunity of view adaptation.

The description motivates the development of complex interactive catalogs by composition of elementary patterns. A detailed formulation of the candidate patterns mentioned above can be found in Appendix A. We will employ them at a slightly generalized perspective:

Search Space Adaptation: generalizes pattern [Selectable Search Space] wrt. categories. It might provide multiple sorts of categories at the same time as, for example, according to time, location, topic, etc. The user may successively adapt his/her selections which may yield a refinement as well as an enhancement of the currently considered sub-space. In addition, the number of available categories is not limited. It thereby comprises the specification form-based approaches as well as keyword-based search.

Set-Based Navigation: is considered in a generalized way by extending permitted user activities. Besides scrolling, it allows activities of changing the visible view onto the list which, in particular, includes adaptation of the order of display and fine-grained enhancement/reduction of the current view.

Interaction History: is considered as proposed in [Tid98] (cf. Appendix A.3).

We then specify interactive catalogs as composite ui-components based on these three patterns. For this purpose, we specify ui-components at a rather abstract perspective. They are denoted by the same name as the respective patterns. Their composition is illustrated in Figure 4.32. Thereby, we first compose ui-components 'Search Space Reduction' and 'Interaction History' which provide a composite ui-component we call 'Search Space Reduction with Backtracking'. A subsequent composition with ui-component 'Set-Based Navigation' then provides an interactive catalog.

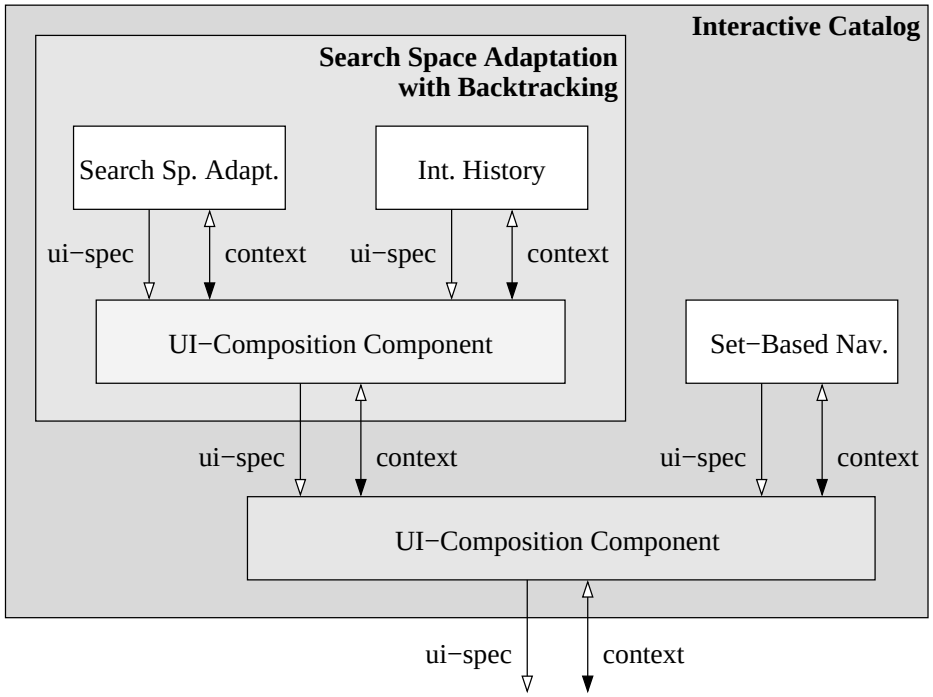


Figure 4.32: Compositional structure of an interactive catalog

According ui-components are realized as follows. Thereby, 'Search Space Adaptation' represents a generalization of 'Category Selection', and 'Set-Based Navigation' represents a generalization of 'List Scrolling'.

UI-Component: 'Search Space Adaptation'

We assume that the underlying 'space' is represented by a base set of elements. The current sub-space is then determined by a function which given a user constraint, it provides a subset of the base set. Figure 4.33 provides an according graphical abstraction. To be general, we define the type of user input as character strings of arbitrary length. It permits, for example, to represent any user input to forms by help of an according encoding (or transformation).

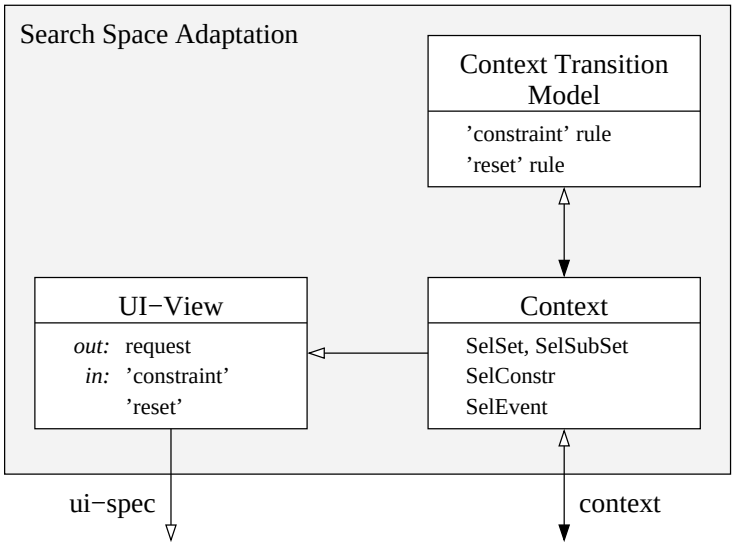


Figure 4.33: UI-Component 'Search Space Adaptation'

Context:

SelSet: represents the underlying set of elements.

SelCat: represents the constraint currently currently provided by the user, or constant 'null' if no restriction is specified.

SelSubSet: represents the subset associated with the currently chosen constraint *SelConstr*.

SelEvent: represents the constraint specified by the user or a 'reset' event.

UI-View: At a basic level, it provides (i) the output to the user and (ii) the activated user events as follows:

Output: Request to the user.

Activated events: Event 'SelEvent' is activated as a string input. If a sub-space is currently selected, a 'reset' event is provided additionally which permits to reset user's scope to the initial space.

Context Transition Model: The context is adapted depending on the constraint specified by the user. An according rule which recomputes subset *SelSubSet* according to the constraint specified.

'constraint' rule: Recompute sub-space.

```
ON updated(SelEvent)
DO {
   update(SelConstr) by SelEvent ;
   update(SelSubSet) by restrict(SelSet, SelConstr)  }
```

where '*restrict*()' denotes a function which determines subset *SelSubSet* depending on constraint *SelConstr*.

'reset' rule: Select original space.

```
ON updated(SelEvent)
IF (SelEvent == 'reset')
DO {
   update(SelConstr) by 'null' ;
   update(SelSubSet) by SelSet
   }
```

There exist two alternatives to represent function *restrict*() used in the 'constraint' rule:

Autonomous approach: To be general, function *restrict*() is realized by a random function. A concrete specification of this function then corresponds to a component refinement (cf. Definition 3.13 on page 66).

Open approach: The specification of function *restrict*() is required as a separate component. As a consequence, ui-component 'Search Space Reduction' provides a further interface. To enable an analysis of the composite component in the case where no *restrict*() function is specified, a general component specification of function *restrict*() is provided. It corresponds to a random function as used above. However, the task of specifying a refined version of 'Search Space Reduction' is significantly relieved. It simply corresponds to providing a refinement of the *restrict*() component.

Note that we base the specification of abstract ui-components on general data types as, for example, character strings. Thereby, component adaptation according to more practical representation types, is achievable by interface refinement (cf. Definition 3.14 on page 67). Thereby, the refinement relation can be derived by specifying an according representation translation.

UI-Component: 'Interaction History'

We apply a straightforward representation of interaction histories in terms of sequences of events. In fact, more general versions are imaginable based on graph representations. Figure 4.34 provides an according graphical abstraction of ui-component 'Interaction History'.

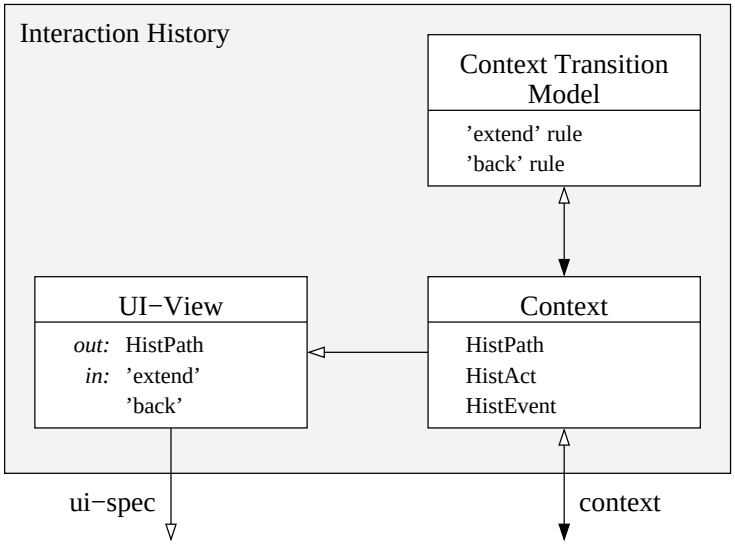


Figure 4.34: UI-Component 'Interaction History'

Context:

HistPath: represents the sequence of events occurred.

HistAct: represents a user activity which should be 'recorded' by the interaction history.

HistEvent: represents the initiated user event according to this ui-component. It comprises any element listed in the current history path *HistPath*.

UI-View: At a basic level, it provides (i) the output to the user and (ii) the activated user events as follows:

Output: History path *HistPath*.

Activated events: Each element of *HistPath* is activated as user event. (Note that for later identification, each event must be internally specified by its position in the list and optionally by its name.)

Context Transition Model: The context is adapted, if a user activity is published by another component through *HistAct*. An according rule then extends the history path by this activity.

'Act' rule: Append an activity to the history path.

```
ON updated(HistAct)
DO update(NavVisSize) by append(HistPath, HistAct)
```

where function $append()$ appends an element to the end of a list.

'Event' rule: Backtrack to a previous activity.

```
ON updated(HistEvent)
DO update(HistPath) by cut(HistPath, HistEvent)
```

where function $cut()$ truncates a list at a specified position.

UI-Component: 'Set-Based Navigation'

The underlying model of 'Set-Based Navigation' resembles that of 'List Scrolling'. It basically provides an extended functionality wrt. user facilities. Figure 4.35 represents an according graphical abstraction.

Context:

NavList: represents the underlying list of elements which may be browsed.

NavListPos: represents the current list position (starting at 1).

NavVisSize: represents the number of list elements presented to a user at the same time.

NavVisInc: represents the number by which *NavVisSize* is enhanced/decreased, if an 'enhance' or 'decrease' event was initiated.

NavOrder: represents the order criteria which is currently applied to *NavList*.

NavEvent: represents the initiated user event according to this ui-component. It comprises events: 'next', 'prev', 'first', 'last', 'enhance', 'decrease', 're-order[]'.

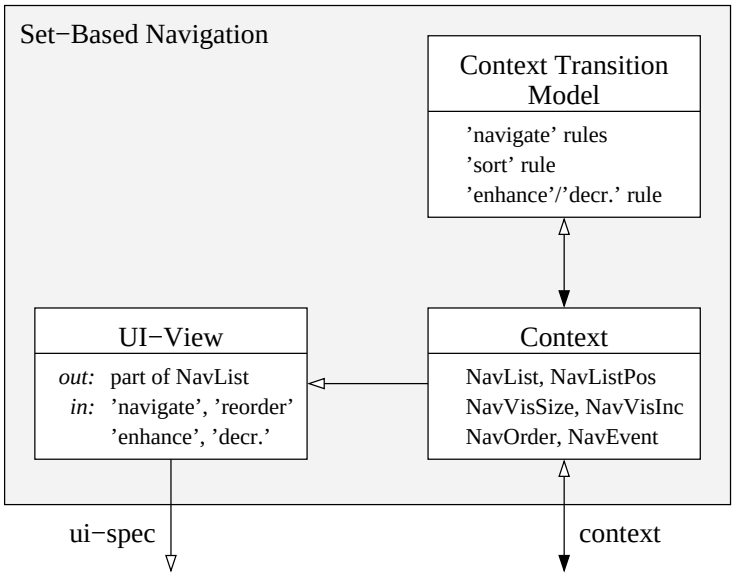


Figure 4.35: UI-Component 'Set-Based Navigation'

UI-View: The specification of (i) the output to the user and (ii) the activated user events are generated as follows:

Output: Sublist of *NavList* starting at position *NavListPos* and comprising *NavVisSize* elements. If the sublist exceeds the end of *NavList*, it is shortened accordingly.

Activated events: Depending on the context, user events 'next', 'prev', 'first', 'last', 'enhance', 'decrease', 'reorder[]' are activated:

'next', 'last': are activated, if

$$NavListPos + NavVisSize \leq length(NavList)\,,$$

where $length()$ provides the number of elements of a list. Thereby, event 'next' corresponds to navigating to the succeeding part of the list, and event 'last' corresponds to jumping to the last part of the list — as specified by the context transition model.

'prev', 'first': are activated, if

$$NavListPos > 1\,.$$

Thereby, event 'prev' corresponds to navigating to the preceding part of the list, and event 'first' corresponds to jumping to the first part of the list.

'enhance': is activated, if

$$NavVisSize \leq length(NavList)\,.$$

The event provides an enhancement of the currently presented view.

'decrease': is activated, if

$$NavVisSize > 1\,.$$

The event provides a reduction of the currently presented view.

'reorder[]': is always activated. This generic event provides a reordering of the currently presented view depending on the particular order criteria selected.

Context Transition Model: The context is adapted depending on the user event initiated. According rules essentially adjust the list pointer wrt. the user event. They are specified as follows:

'first', 'last', 'next', 'prev' rules: They correspond to according rules of ui-component 'List Scrolling' introduced in Section 4.2 (on page 82).

'enhance' rule: Enhance visible view.

```
ON updated(NavEvent)
IF ( NavEvent == 'enhance' )
DO update(NavVisSize) by NavVisSize + NavVisInc
```

'decrease' rule: Shrink visible view.

```
ON updated(NavEvent)
IF ( NavEvent == 'decrease' )
DO {
   IF ( NavVisSize > NavVisInc )
     update(NavVisSize) by NavVisSize - NavVisInc
   ELSE
     update(NavVisSize) by 1
   }
```

'reorder[]' rule: Reorder element list.

```
ON updated(NavEvent)
IF ( NavEvent == 'reorder[]' )
DO {
   update(NavOrder) by reorder[] ;
```

```
update(NavList) by sort(NavList, reorder[])
}
```

where *sort*() denotes a function which computes the reordered list based on the criteria selected by the user. According to function *restrict*() used at 'Search Space Reduction', *sort*() may be realized by a specific component.

As we intend to employ 'Set-Based Navigation' within dependent compositions, we consider an open approach. We explicitly admit that context 'NavList' may be manipulated by other ui-components (e.g., 'Search Space Reduction'). For this occasion, the following 'repairing rule' is applied. It resets the list position and reorders the list according to the currently selected order criteria.

'List' rule: Reset list position and adapt list order.

```
ON updated(NavList)
DO {
   update(NavListPos) by 1 ;
   update(NavList) by sort(NavList, NavOrder)
   }
```

UI-Composition component:
'Search Space Reduction' + 'Interaction History'

The intention of this composition is to provide a backtrack facility for ui-component 'Search Space Reduction'. There exist dependencies in both directions. (i) If an event wrt. 'Search Space Reduction' is initiated, it should be appended to the interaction history. (ii) If a backtrack event wrt. 'Interaction History' is initiated, a corresponding previous dialog situation must be re-established.

Context: Contexts *SelSubSet*, *SelConstr*, *SelEvent*, and contexts *HistAct*, *HistEvent* are assumed to be declared as public by ui-components 'Search Space Adaptation' and 'Interaction History' respectively. A further context extension is not required.

UI-View Composition: Interaction composition is provided by the union of (i) the user outputs and (ii) the activated events of both ui-components.

Context Dependency Model: The dependencies explained above are realized by two rules:

Publish rule: Publish event to interaction history.

```
ON updated(SelEvent)
DO update(HistAct) by SelEvent
```

'Backtrack' rule: Re-establish a previous state.

```
ON updated(HistPath)
DO update(SelConstr) by HistPath[length(HistPath)];
```

Note that we assume an open perspective. Thereby, an update on context *SelConstr* initiates a repairing rule of ui-component 'Search Space Reduction' which accordingly adapts its context. In particular, it has to recompute context *SelSubSet*. An according rule reads as follows:

```
ON updated(SelConstr)
DO update(SelSubSet) by restrict(SelSet, SelConstr)
```

The composition of the ui-composition component together with ui-components 'Search Space Reduction' and 'Interaction History' then provides a composite ui-component *'Search Space Reduction with Backtracking'*.

UI-Composition component:
'Search Space Reduction with Backtracking' + 'Set-Based Navigation'

The intention of this composition is to provide an interactive catalog to users. They may specify a sub-space of interest through an iterative refinement process. If a desired level of detail is reached, they may browse the selected sub-space or backtrack to previous situations.

There exists a (directed) dependency from 'Search Space Reduction with Backtracking' to 'Set-Based Navigation'. If a sub-space is selected by the user, the according element list provided by 'Set-Based Navigation' must be adapted. The dependency corresponds to dependency between 'Category Selection' and 'List Scrolling' discussed in Section 4.3.1 (on page 88).

Context: Contexts *SelSubSet*, *SelEvent*, *HistEvent*, and contexts *NavList*, *NavEvent* are assumed to be declared as public by ui-components 'Search Space Reduction with Backtracking' and 'Set-Based Navigation' respectively. A further context extension is not required.

UI-View Composition: Interaction composition is provided by the union of (i) the user outputs and (ii) the activated events of both ui-components.

Context Dependency Model: The dependency explained above is realized by the rule:

```
ON updated(SelConstr)
DO update(SelSubSet) by restrict(SelSet, SelConstr)
```

The composition of the above ui-composition component together with ui-components 'Search Space Reduction with Backtracking' and 'Set-Based Navigation' then provides a composite ui-component *'Interactive Catalog'*. Thereby, 'Interactive Catalog' provides according dialog structure at an abstract level. At a practical perspective, several adapted versions of 'Interactive Catalogs' should be provided. Commonly, according adaptations will correspond to refined versions of ui-component 'Interactive Catalog' in terms of component refinement. While behavior refinement will not be achieved usually, interface refinement is achievable mostly. Thereby, the following selected versions of ui-component 'Interactive Catalog' represent refinements of practical relevance:

Multi-dimensional categories: While ui-component 'Category Selection' provides a one-dimensional category, common catalog structures are based on multi-dimensional categories. To refine a search, a user selects a category from a chosen dimension. It corresponds to drill-down facilities of OLAP interfaces (cf., for example, [AGS97]). It can be achieved by refining the specification of sub-component 'Search Space Reduction'.

Predefined refinement scenarios: A common interaction style to support a user in searching for items in a catalog is provided by predefined scenarios. At each step a user may select a category wrt. a particular dimension. It corresponds to a further refinement of multi-dimensional categories.

Keyword-based search: Keyword-based search is realized by a refinement of sub-component 'Search Space Reduction' as well. Thereby, the refinement particularly concerns its function 'restrict()'. Depending on the keyword(s) chosen by the user, function 'restrict()' computes an associated subset.

If required by the application, ui-component 'Interactive Catalog' can be used for more complex compositions in turn. A typical example is the composition of an interactive catalog within commercial applications providing products. Thereby, ui-component 'Interactive Catalog' will be composed, for example, with a ui-component representing interactive facilities of a shopping cart.

A simplified net-based specification of a single-level catalog is proposed in [Hei01]. For its simulation, an environment based on a HTML driver is developed. It permits a Web-based navigation which generates the underlying list by querying a connected database. Thereby, it applies a variant of CP nets introduced in [Lew00].

4.7 Concluding Remarks

Regarding context specification, we permit a distinction between private and public contexts which supports encapsulation. It might be advantageous to include further distinctions. For example, attribute ”public” does not state whether a ui-component actually manipulates this context or only retrieves its state. A distinction between ”read-only”, ”write-only”, and ”read-write” can be used, for example, to derive desired properties from the underlying rule system. An according discussion about issues of ECA rules is presented in the following.

To specify dependencies between ui-components, we applied the method of ECA rules. As shown at the examples, they provide a powerful tool for this purpose. Throughout the thesis, we generally followed a rather unrestricted perspective wrt. expressiveness. While expressiveness is naturally desired, there commonly exist trade-offs between expressiveness and properties which can be guaranteed. Regarding ECA rule systems, interesting properties are known as *termination*, *confluence*, and *effect preservation*. Thereby, the termination property guarantees that a specified rule system always terminates independent from the state and the initiating event. The confluence property guarantees that for any selected state and any selected event, the rule system always reaches the same final state, or it does not terminate at any case. The property of affect preservation guarantees that a performed update is not invalidated by the rule system (for example, by ”accidentally” reversing the affect of an update). These properties have been studied intensively at the database area (cf., for example, [ST98, ST99, BJ02b, BJ02a, BDR98, AHW95, BW00]). In particular, according restrictions have been identified to guarantee particular properties. They can be applied to our work, if we consider the rule system at a global perspective. More precisely, according restrictions have to be applied to the rule system which comprises all rules specified at any level.

However, according to a component perspective, it would be beneficial to verify restrictions at a local perspective. While we outline an approach regarding termination in the following, we currently consider a localized treatment of confluence and effect preservation as open work. According to termination, we consider a ui-component (or ui-composition component) $\mathscr{U}$ at an arbitrary level L. As a necessary requirement for termination, we have to verify that the local rules together with all lower-level rules terminate. In general, termination is guaranteed if no cycle occurs in the associated dependency graph. Thereby, nodes of the dependency graph represent the single contexts. A directed arc (c_1, c_2) belongs to the graph, if there exists a rule where c_1 occurs in its event part and c_2 is updated at its body. As a necessary requirement, we have to ensure that the local rule system does not possess any cycle. At this step, there is no difference to the general case. However, even if there are no cycles locally, a cycle might be accomplished by rules defined at a lower level. It can occur, if (i) a rule at level L updates a context c, and c occurs in the event part of a rule at a lower level, *and* (ii) a rule at a lower level updates a context c', and c' occurs in the event part of a

rule at level L. Note that a cycle can only be accomplished by lower level rules, if *both* conditions are valid for some contexts c, c'. Thus, cycles are generally avoided if one of the above conditions can be falsified for any context c. Note that this verification can be performed locally without considering the complete rule system. To guarantee termination at the global level, any ui-component needs to verify termination locally.

For example, consider the context dependency between 'Category Selection' and 'List Scrolling'. We consider the termination of the rule system of the ui-composition component which connects them. It only possess the single 'SelSubSet' rule. Firstly, context 'SelSubSet' occurs in the event part of the 'SelSubSet' rule. Because sub-component 'Category Selection' updates context 'SelSubSet', condition (ii) is valid. However, condition (i) is invalid wrt. 'Category Selection', since there does not exist a context which is updated by the 'SelSubSet' rule and which occurs at the event part of a rule of 'Category Selection'. Secondly, context 'NavList' is updated by the 'SelSubSet' rule. Because 'NavList' occurs at the event part of sub-component 'List Scrolling', condition (i) is valid. However, condition (ii) is invalid wrt. 'List Scrolling', since there does not exist a context which occurs in the event part of the 'SelSubSet' rule and which is updated by a rule of 'List Scrolling'.

Chapter 5

Related Work

The introduced approach relates to several research areas. As our work was initially motivated by the formalization of Web information services, we will provide an overview of existing research approaches at this area in Section 5.1. Although they particularly concern Web interfaces, their main goals are closely related to our work. They commonly provide formal specifications to support service quality, and partially propose opportunities of reuse. In particular, they explicitly support user interaction design by formal specifications.

Currently, component approaches are a promising area of research. In the thesis, we applied a particular component approach based on stream relations [BS01]. A wider perspective is presented in Section 5.2.

We omit a discussion about existing specification models for user interaction. As motivated in the introduction, there exist many established specification models. Good overviews are provided, for example, in [DFAB98, Lew00]. According to place/transition systems, models of composition are introduced, for example, in [Bau96, BDK01]. They commonly permit rather general versions of composition. We proposed a particular and restricted model of net composition wrt. CP nets. In contrast to the above, this composition model permits (i) a characterization of the external behavior, (ii) desired behavioral properties of the composition, and (iii) an embedding into the component framework introduced in [BS01].

Throughout the thesis, we applied the notion of interaction patterns at a common perspective. Thereby, we focused on an agreed core and neglected peculiarities of the different approaches [Bor01, Tid98, WT00a, WT00b, DLH02, GPBV99, RSL00]. For example, [Bor01] extends the structure of interaction patterns by an attribute called *ranking* adopted from [Ale79]. It indicates the confidence of the validity of an interaction pattern. In particular, if a pattern has been applied successfully within different contexts, a higher ranking is associated with the pattern. We omit a detailed discussion of these approaches, since their differences do not affect the general intentions.

5.1 Design Models for Web Information Services

Several approaches to designing Web information services have been proposed in the last years. Besides design methods and specification models, they commonly provide an opportunity to execute final design results in terms of executable Web applications. Depending on the approach, either appropriate mappings of design specifications into implementations are provided, or particular management systems are developed which interpret declarative specifications. Approaches which primarily focus on Web-site management systems are omitted in the following, as they are not as closely related to our work. According approaches are introduced, for example, in [FLM98, FFK+98, MAG+97].

In the following, we briefly introduce the following approaches:

Araneus: a Web design methodology which combines design models of relational database design with design models of hypermedia design (proposed by Atzeni et. al. [MAM03, AMM98]),

OOHDM: an object-oriented hypermedia design method which adapts a UML approach to hypermedia design (proposed by Schwabe et. al. [SR98, RSL99, SREL00]),

Torii: a data-driven approach and tool environment to specification and automatic generation of data-intensive Web applications (proposed by Ceri et. al. [CFP99, FP98]),

WebComposition model: a component-oriented approach to web-site design (proposed by Gaedke et. al. [GG00, GGS+99a, GGS+99b]),

View-centered design model: a design model based on generalized database views which permits user and device adaptation (proposed by Thalheim et. al. [FST98, FKST00, TBF+98]).

Generally, these models follow rather formal approaches. In addition, a facility of composition is supported by the WebComposition model and a facility of verification is provided in [GC99] according to OOHDM. However, in comparison to the approach introduced in this thesis, non of the above approaches provides both facilities: (i) compositionality and (ii) verifiability.

5.1.1 Araneus

Araneus [MAM03, AMM98] is a Web design methodology which is supported by design tools. It provides a data-centered design method, since it proposes to start the design process by an Entity-Relationship schema (ER schema). This schema is used

as a basis for the following design steps. As illustrated in Figure 5.1, the design model distinguishes between the design of data structures and the design of hypermedia structures. Thereby, data structures are evolved by a common database schema refinement process. It eventually results in a relational database schema. The design of hypermedia structures is supported by a refinement process which is divided into three major transformations. Each transformation results in a more refined specification. Thereby, the ER schema is used to derive an hypertext conceptual schema (NCM). It basically describes the hypermedia structure at an abstract level. It specifies the navigation structure on concepts rather than on concrete pages. Thereby, a connection to the conceptual database schema is realized by associating concepts of the NCM schema with types and attributes of the ER schema. Subsequently, a hypertext logical schema is derived. It contains the actual page structure and an associated navigation graph. In addition, a connection to the logical database schema is realized by using attribute names for the specification of page schemes. Finally, the concrete layout of single pages is specified by using page templates or a transformation language as XSL. Thereby, a case tool called 'HOMER' thoroughly supports the design process. A management system called 'PENELOPE' can be used (i) to generate a web-site from the resulting specification (compiler approach) or (ii) to execute the specification (interpreter approach).

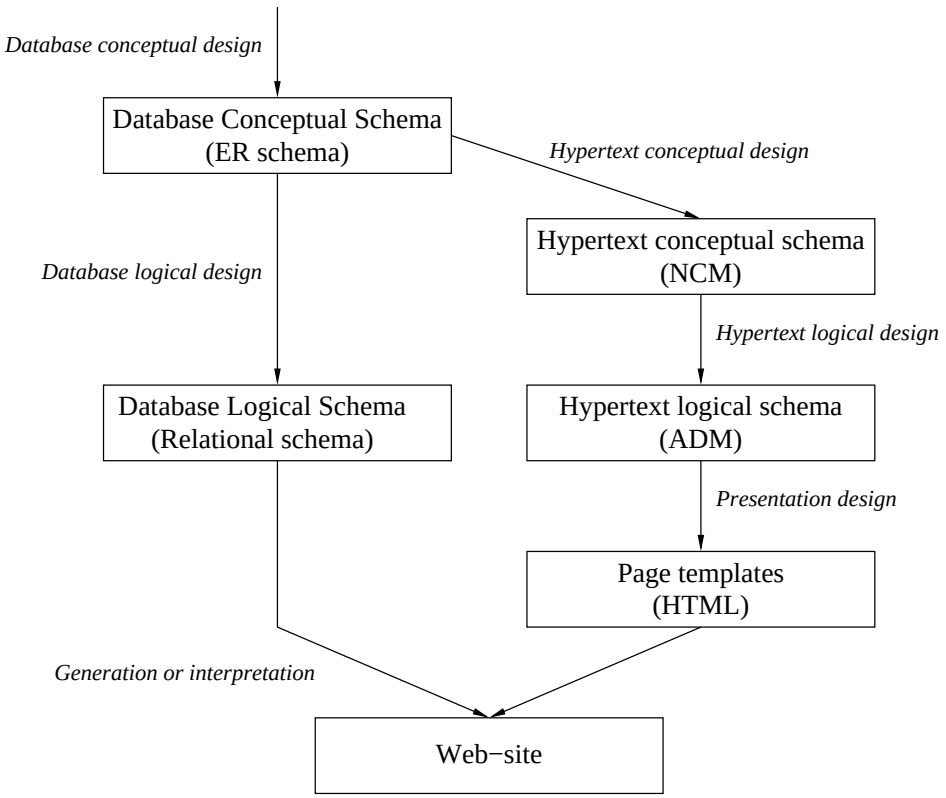


Figure 5.1: Araneus design process

5.1.2 OOHDM

OOHDM [SR98, RSL99] is an object-oriented hypermedia design model which adapts an approach based on UML to hypermedia design. It distinguishes four primary design activities: conceptual design, navigation design, abstract interface design, and implementation. In comparison to an interface independent UML approach, their distinguishing features are:

- a notion of *navigation objects* which represent views on conceptual objects (comparable to the notion of database views),
- a notion of *navigational context* which provides an abstract specification of the navigation space, and
- the separation of issues of layout from navigational issues.

Their major design steps resemble that of the Araneus approach. As a fundamental difference, Araneus starts from a conceptual data-oriented schema, and OOHDM start starts from a conceptual object-oriented schema. In the following, we briefly explain their four primary design activities.

Conceptual design: The conceptual modeling corresponds to the UML approach. Peculiarities of Web interfaces are not yet considered at this design step. Its main design result is a (slightly extended) class diagram. However, other opportunities of UML as, for example, use case diagrams may be applied additionally.

Navigational design: At this design step, the navigation structure of an application is developed. It is realized within two steps which logically correspond to the design steps of ”Hypertext conceptual design” and ”Hypertext logical design” of the Araneus approach. Firstly, a so-called *navigational class schema* is derived from the conceptual schema. It represents a view on the conceptual schema. Derived navigational class schemes thereby reflect a single or multiple user’s perspectives onto the conceptual schema. To semantically characterize navigational classes (i.e., classes of the navigational class schema), there exist predefined types: node, link, and access structure. Roughly, nodes represent later pages, links represent a relationship between nodes (interpreted as navigational paths), and access structure as, for example, indices and guided tours represent possible ways of accessing nodes. In particular, nodes reflect object-oriented views on conceptual classes specified by a query language. For example, they may combine attributes from different conceptual classes. Links and access structures define navigation paths in abstract terms.

At a second step, a so-called *navigational context schema* is derived from the navigational class schema. It further structures the navigation space. Thereby,

nodes are associated with *contexts*. A context then further characterizes a node by means of properties (e.g., only represent books of a certain category) and navigational opportunities. This refinement step particularly permits that the visualization of nodes automatically adapts at runtime depending on the current context within a user scenario.

Abstract interface design: After the navigation space is structured, the layout of pages is defined in abstract terms by the abstract interface design. In particular, this step does not include concrete HTML code specifications. For example, it associates attributes with layout definitions which specify where attributes should be placed relative to others. In addition, it defines which objects permit user interaction.

Implementation: The final implementation concerns the realization of information items, context, and the user interface. Commonly, conceptual and navigational objects are mapped into an according database structure. The dynamic notion of contexts must be realized by an interpreter which particularly computes according database views.

Web design frameworks [SREL00], a slight extension to OOHDM, introduce a restricted opportunity to reusing navigational specifications. It extends the approach by so-called *hotspots* known from the framework area. Hotspots characterize flexible elements which may be instantiated by the designer according to application requirements. For example, they introduce a hotspot called ”generic context”. It represents a flexible element of the navigational context schema which may be instantiated by a concrete navigation context. Thereby, reuse is rather restricted to specifications of the same application domain, since the navigation schema is associated with a particular conceptual schema.

Independent from the group of Schwabe et. al., Germán et. al. [GC99] proposed a formal specification language which permits a representation of OOHDM design specifications by a high-order logic. Therewith, they enable verification support for OOHDM designs.

5.1.3 Torii

Torii [CFP99, FP98] provides a data-driven approach and tool environment for the specification and generation of data-intensive Web applications. As a distinguishing feature, the Torii approach thoroughly deploys a XML syntax for the representation of design specifications at all levels. Torii design is structured by the following design levels which resemble those of the Araneus approach.

Structural model: describes the data structures at the conceptual level by a slightly extended version of the Entity-Relationship model. The extension permits the

additional notion of ”target” which specifies an aggregation of elementary concepts. Besides targets specified by a designer, Torii offers two predefined targets which commonly apply to Web information services. These are called ”Profile” and ”Metadata”. While target ”Profile” contains information regarding users and user groups, target ”Metadata” contains meta information as, for example, modification and usage patterns, and access restrictions. According to a particular design, these targets can be adapted by specialization.

Derivation model: specifies conceptual views. The designer derives different views from the conceptual schema according to the various tasks and users of the targeted application. Since specifications at all levels are based on XML, views are derived from the structural model through an XML query language.

Composition model: specifies the content of each page. By default, each concept of the derivation model corresponds to a single page. To adapt this default, the designer may associate single pages with attributes of several concepts or, conversely, associate single concepts with several pages. Thereby, different assignments may be defined for different users.

Navigation model: defines the navigation structure. As a default, Torii assumes that links between pages are commonly derived from the semantic relationships defined at the structural model. It is referred to as *contextual navigation*. In contrast, *non-contextual navigation* denotes links between pages which are conceptually unrelated. To represent a ”one-to-many navigation” (i.e., one object is related to several objects), *navigation modes* are specified by determining several semantic characteristics of the relationship. For example, it specifies the number of objects presented on the same page, their order, and the mode of browsing as, for example, forward/backward scrolling. Instead of defining particular navigation modes, predefined navigation modes may be selected. They comprise, for example, index navigation, guided tours, or ”show all” — concurrently presenting all objects.

Presentation model: The page layout is defined on the basis of XML transformation rules. They specify the transformation of page specifications to HTML. To permit prototyping, a default page style is provided based on a simple layout.

Business rules: As a second distinguishing feature, Torii provides so-called business rules. They are primarily deployed for user adaptation. They are defined in terms of event-condition-action rules. Events are user events (i.e., the invocation of another page) or data changes. Conditions evaluate predicates or database queries. Finally, actions comprise adding or dropping elements from collections, initiating database updates, sending emails, or assigning users a selected site view.

Torii is supported by the following tools. At the design layer, modeling tools are provided. Their design results are subsequently transformed by a pre-runtime processor into an intermediate representation. It is adapted to the requirements of an performant execution. At the runtime layer, an interpreter executes the transformed specification. Thereby, it initiates according database requests, merges retrieved data with the page templates, and finally computes the transformation to HTML pages.

5.1.4 WebComposition Model

The WebComposition model [GG00, GGS+99a, GGS+99b] is a component-oriented approach to web-site realization. Its primary intention is to support reuse and maintenance of web-sites. The proposed architecture provides management of a component repository on the one hand and a document generator on the other. Thereby, the management system supports the development and retrieval of components. The document generator interprets composite components by iteratively assembling outputs (i.e., fragments of HTML code) of associated sub-components.

A component is basically specified by a list of properties and operations. Thus, components are comparable to objects at an object-oriented perspective. While properties represent a static view onto components, operations provide the dynamics. The most essential operation each component is obliged to implement is called 'generateCode()'. This operation specifies the visualization of a component at a certain state. More precisely, 'generateCode()' computes a fragment of HTML code on request. Thereby, components correspond to a model/view paradigm.

Two types of relationship are supported by the WebComposition model: inheritance and aggregation. Inheritance can be compared with inheritance of classes. It permits to introduce new properties and operations and to overwrite properties and operations inherited from a component. Note that components are thereby considered in a dual sense — as classes and as objects. The aggregation relationship provides two opportunities. Firstly, a component aggregated of sub-components may represent their cumulative output, i.e., an HTML page which is assembled of HTML code fragments generated by each sub-component. Secondly, aggregation of components may represent hypertext links. Besides sequential composition of HTML fragments and linkage of HTML pages, there is no indication how more sophisticated dependencies can be specified.

Generally, the WebComposition approach exhibits a close relationship to the approach introduced in the thesis. However, it falls short in respects of (i) user interface independence, (ii) verifiability, and (iii) dependency specification.

5.1.5 View-Centered Design Model

In [FST98, FKST00, TBF+98], a design model based on generalized database views is proposed. As a distinguishing feature, it provides opportunities for both user adaptation as well as device adaptation. It is mainly based on the notions of *information units* and *information containers*. They can be related to the Araneus approach introduced above. Roughly, information units correspond to elements of the hypertext conceptual schema (NCM), and information containers correspond to elements of the hypertext logical schema (ADM). In contrast, information units extend conceptual views by abstract style information and standard functionality used in Web information services. Information containers extend page specifications by by meta information required for device adaptation.

Information units

Based on a higher-order Entity-Relationship schema [Tha00, Tha93] (HER schema), information units are derived in terms of generalized views. They result from three basic transformations: filtration, summarization, and scaling. Filtration derives a view on a HER schema in the database sense. It is derived by applying schema transformations which primarily include omitting types and attributes, composing new types, and decomposing existing types. By summarization, new attributes may be derived based on existing ones. It is defined in terms of queries over the schema. Scaling provides an opportunity of customization. It specifies measure rules, ordering rules, adhesion rules, and hierarchy meta rules. Measure rules specify the conversion of prices, weights, etc. Ordering rules specify the ordering of objects wrt. user presentation. By associating weights, adhesion rules further specify which objects should be presented together, and which objects might be separated. It provides information for a later device adaptation. For example, devices with small displays might require to split information onto several pages. Thereby, less associated objects may be represented on different pages. Hierarchy meta rules specify different abstract views onto objects. According to device adaptation, more compact representations may be selected, if visual capabilities of a device are limited.

Besides static semantics of information units, they are associated with interactive facilities which provide standard functionality used in Web information services. Examples are generalization and specialization which adapt the level of detail of presented information (comparable to role up and drill down facilities of OLAP interfaces [AGS97, GL97]), reordering which adapts the arrangement of information (comparable to pivoting facilities of OLAP interfaces), and browsing functions which permit navigation within information units.

Information containers

As an extension to pages, they represent abstract interaction specifications. At a concrete device, they may correspond to a single page or several pages with an appropriate

navigation structure. The container metaphor distinguishes between an an ”unloaded” and a ”loaded” state of a container. At the former, a container is specified by certain restrictions which reflect limitations of capabilities of a device. These restrictions are specified in terms of parameters. For example, they define size and type of displays. At runtime information containers are instantiated based on information units. Thereby, associated restrictions are considered, for example, by computing possible page splittings, or representing objects by more compact alternatives.

5.2 Component Approaches

In the thesis, we primarily focused on issues of formal specification and composition of components. Thereby, we less emphasized on associated issues of the design process. Besides, relating the applied component model to other work, we provide a survey about design-related aspects.

5.2.1 A Design Perspective

Component approaches are a topical area of research [LS00, BS01, Szy98, BDH+98, Fra99, AN01, FT02a], although according visions are proposed already decades ago (cf, for example, [McI68]). The concept of design by components is essentially motivated by two aspects: quality and reuse of designs. Thereby, reuse directly implies an improved efficiency of the design process. As a discussion provided in [BDH+98] among several known researchers of the component area manifests, there is no common agreement about a delimitation (or even definition) of the notion of components. For example, the following distinct component definitions are proposed in [BDH+98]:

> K. Koskimies: ”A component is a system-independent binary entity which implements one or more interfaces. An interface is a collection of signatures of services belonging logically together.”
>
> C. Szyperski: ”A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

Although there does not exist neither an agreement about a particular component definition, nor a form (or specification) by which a component should be provided, there are certain characteristics of components which are widely agreed upon. They comprise

Reusability: The requirement of reusability is the most significant. It partially implies the following properties.

Compositionality: Components can be composed iteratively. Thereby, it is expected that the composition is reliable in the sense that it guarantees particular intentions.

Interface specification: Interfaces are provided as a basis for composition. Commonly, they represent a static notion of composition, for example, specified by signatures on data types.

Adaptability: Regarding the deployment of components within different environments, opportunities of adaptation should be provided. Besides deployment, adaptability supports maintenance of applications.

Commonly, formal approaches also consider system independence as an important feature of component models.

Generally, there is a significant difference between a particular component model and an according component-based design method. In [Fra99], the following requirements are identified to permit a ”design by components”:

- support for finding desired components within a repository,
- support for semantic understanding of components (wrt. a designer's perspective),
- facilities for flexible and reliable component composition, and
- facilities for component adaptation and substitution.

In detail, the following approaches to their support are proposed. Retrieval of desired components can be supported by syntactically searching through source code, interface signatures, component names, or verbal descriptions as far as they are provided. Thereby, thesauri may be used as a further supporting structure.

To support semantic understanding of components, formal specifications are usually not sufficient at a designer's perspective. Therefore, it is proposed to associate components with verbal descriptions possibly enriched by diagrams. However, as the structuring of descriptions strongly influences their comprehension, a suitable and unified structure must be identified and imposed on authoring. As an opportunity to its realization, [Fra99] proposes to apply design patterns for this purpose. In particular, design patterns of the style proposed by Gamma et. al. [GHJV95] provide a promising candidate, since they are accepted to a certain extend. The underlying motivation is that design patterns provide both a unified textual structure together with diagrammatic illustrations. In addition, [Fra99] proposes to use according formulations for supporting component retrieval as well. Note that this approach strongly corresonds to our proposal. To identify desired ui-components, firstly an identification of according

interaction patterns supports a designer at the tasks of retrieval as well as comprehension.

According to composition support, different modes of component provision are suggested: black-box, glass-box, and white-box. While black-boxes hide internal information, glass-boxes permit a view onto the internal structure, and white-boxes permit a direct adaptation of component specifications. The choice of an according mode usually corresponds to a trade-off between flexibility and reliability. Thereby, reliability concerns the correctness of possible compositions. While a white-box deployment of components provides maximum flexibility, black-box deployment commonly promises maximum reliability. At a practical point of view, the adaptation of white-box components may result in an unexpected behavior of compositions — thus, degrading reliability. To guarantee reliability of (black-box) compositions, the notion of *contracts* was proposed (cf., for example, [SN99, Szy02]). A contract specifies the constraints imposed onto an interface. More precisely, it defines dynamic obligations components connected at a particular interface have to obey. It is interpreted as: if a client component satisfies its part of the contract, then the provider guarantees a particular behavior (in terms of service provision). Although, we did not explicitly adopt the notion of contracts in the thesis, it corresponds to the treatment of properties and assertions introduced in Section 3.3.2. At the perspective of a service provider, assertions correspond to that part of a contract, a client component has to satisfy. If satisfied, the provider guarantees particular (dynamic) properties.

According to support (black-box) adaptation, commonly a notion of refinement or inheritance is introduced. While rather object-oriented component approaches basically define inheritance in terms of redefining data types of interfaces, other approaches define refinement in terms of the dynamic behavior (cf., for example, [BS01]). On the one hand, the latter approach strongly supports reliability, since it permits to derive behavioral statements about refined components. On the other hand, it is not a trivial task to verify the refinement relationship.

5.2.2 Related Component Models

Besides, the component model proposed in [BS01] applied in the thesis, other models can be found in the literature. According to our concern of formal and system-independent component models, a good survey on this topic is provided in [LS00]. In comparison to [BS01], the model proposed by Nierstrasz et. al. [SN99, AN01] provides the closest relationship. It is based on a formal specification language called $\pi\mathscr{L}$-calculus [LAN00]. In contrast to [BS01], they introduce a significantly richer semantic structuring. Particularly, they propose a layered component approach based on five layers. In bottom-up order, these comprise: the $\pi\mathscr{L}$ abstract machine, the piccola language, core libraries, architectural styles, and applications. They realize the following responsibilities:

$\pi\mathscr{L}$ abstract machine: provides an abstract machine in which *agents* asynchronously communicate *forms* through shared *channels*. Thereby, forms represent generalized tuples. They represent the ”values” which are communicated between agents.

Piccola language: defines syntax and semantics of the composition language. It introduces primitive values, and higher-order abstractions over agents, forms, and channels. Thereby, abstractions correspond to a translation to the lower level model. In particular, forms at this level represent component interfaces, arguments (wrt. provided services), and user-defined services.

Core libraries: define basic composition abstractions. For example, it includes control abstraction as ”if-then-else”, and utilities as an interface to Java.

Architectural style: comprise, for example, streams (at push or pull semantics), GUI abstractions, as well as glue abstractions.

Applications: defines the adaptation and composition of components to applications.

In comparison to our approach, there are two interesting, open questions. Firstly, does the approach permit an analogous embedding of interaction nets? Intuitively, the question might allow a positive answer, since notions of agents, forms, and channels at the elementary layer roughly correspond to notions of component specifications, messages, and channels used in [BS01]. Secondly, since the component model provides a rich semantic structure, it might be possible to directly specify ui-components and their dependent composition by higher-level semantic concepts provided by their model, since they provide a semantic abstraction wrt. the graphical user interfaces. However, an answer to this question has not yet been followed intensively, and is considered as an open problem.

Besides the component model of Nierstrasz et. al., a powerful generalization of the component model of Broy et. al. was proposed in [BRS$^+$00]. It mainly extends the flexibility of the component model. Thereby, concepts which are originally defined as static, are replaced by a dynamic notion. More precisely, the following elements of the model may change during runtime: (i) the set of components deployed for an application, (ii) the assignment of interfaces to components, and (iii) the assignment of connections between interfaces. Thereby, they denote a particular tuple of components, interface assignments, and connection assignments as a *configuration* or instance of the *configuration space*. It particularly increases flexibility, since components may be introduced into and removed from an existing configuration during runtime. It relates to our work, since it permits a more flexible treatment of configurations of ui-components. For example, at sequential compositions (wrt. user interaction), activation and deactivation of ui-components can be specified in terms of

changing configurations. Therefore, such an extended approach requires the specification of configuration adaptation. Intuitively, it might yield more compact and readable specifications.

5.3 Design of Information Services

In the following, we will discuss how a design approach based on ui-components relates to existing design approaches for information services. In particular, we will relate the component-based approach to the design approach based on the storyboard metaphor and to the design approach based on codesign of data structures, functionality, and user interface.

A storyboard perspective

According to the design process of information services the metaphor of a *storyboard* has been successfully introduced and applied [TD03, NL00, DT02]. It associates elements and activities known from storyboard design at the context of theater play or movie production with elements and activities occurring at the design process of information services. In [TD03], storyboard design comprises modeling of actors, dialog scenes, transitions between scenes, dialog steps, and media objects. Thereby, *actors* correspond to the different user groups. *Dialog scenes* represent activities at a non-elementary level. Depending on the abstraction level, composite activities as performing a reservation, accomplishing necessary steps of payment, or processing documents may represent dialog scenes. Commonly, they comprise several steps which are semantically associated to one another. *Scene transitions* represent a change from one scene to another. These changes are initiated by the user and reflect an accomplishment of the preceding scene. In contrast to storyboards in the context of theater play, scenes are not exclusively ordered sequentially. There may exist several alternative continuations. Examples of scene transitions are initiating payment after all desired products are collected, or archiving course information at the end of a semester. *Dialog steps* represent elementary interactive units as, for example, filling out a registration form, or adding a new course to the course schedule of the next semester. Finally, *media objects* represent information views. More precisely, information presented to the user at certain dialog steps is provided through media objects. They can roughly be understood as views over a database including desired style information.

These basic constituents can be identified at the proposed component approach as follows: Media objects roughly correspond to the output specification generated by ui-components. More precisely, output specifications are generated by the ui-view subcomponents by means of context requests. Dialog steps are realized by ui-components. Thereby, dialog specifications generated by a ui-component corresponds to a particular dialog step. Thus, ui-component specifications commonly comprise several dialog

steps. Dialog scenes are represented by composite ui-components. For example, the dialog scene representing an activity of searching and selecting products corresponds to a composition of ui-components [Interactive Catalog] and [Shopping Cart]. Transitions between scenes are basically represented by composition components. By according rules, they determine at which context state a transition to a successive dialog scene may be initiated. Thereby, the current scene is abandoned by deactivating output of associated ui-components, and a successive scene is opened by activating output of associated ui-components. Different characteristics of actors as, for example, rights and preferences, can be represented by the context and context adaptation rules at a global level. Thereby, according rules specify a masking policy according to dialog steps as well as user adaptation.

A codesign perspective

The codesign approach concerns an integrated design of data structures, functionality, and user interaction at all layers of the design process (cf. [BCN92, CT97, Tha00]). In particular, [BCN92] proposes a view-centered codesign approach. A decomposition of the targeted application is identified. Resulting parts of the application are design by means of *generalized views*. Besides data structures, generalized views comprise associated functionality and facilities of user interaction. After the generalized views have been designed, they are integrated iteratively. In fact, ui-components coincide with this design perspective. UI-components can be considered as generalized views. Thereby, data structures are specified in terms of an underlying structure of the context. Functionality and user interaction are specified in terms of context rules and the ui-view. Thereby, a designer may apply a codesign approach to develop elementary ui-components. The integration of generalized views corresponds to component composition. It comprises the integration of all three dimensions which is realized by the specification of ui-composition components. Data integration is specified by (i) context integration, (ii) definition of context dependency rules, and (iii) possible context extensions. Functionality is integrated by context transition rules. Finally, interaction specification is defined by the specification of ”UI-View Composition”. Therefore, the codesign approach may be utilized to support the design of components.

Chapter 6

Conclusions

In the following, we summarize the achievements of the component-based approach introduced in the thesis, indicate open problems, and as far as possible, suggest alternative directions to their solution.

6.1 A Summary

In abstract terms, the thesis proposes a formal component model which permits modular specification, composition, and adaptation of dialog structures. Thereby, it distinguishes between two perspectives (i) the underlying component model and (ii) its utilization according to user interaction. While the underlying model provides the basis for formal specification, composition, simulation, and verification, its utilization in terms of ui-components and their (semantic) composition supports interaction specification at the level of design. At selected examples, it could be shown that dialog structures formulated in terms of interaction patterns can be formally specified by ui-components. As motivated at the introduction, composition of interaction patterns plays an essential role according to the development of pattern repositories and according to their deployment for concrete application designs. To this respect, the notion of ui-composition components (or composition patterns at a generic perspective) was introduced which enables the composition at a semantical level. Thereby, dependencies between patterns are specified explicitly. At exemplary cases, composite interaction patterns were derived in terms of complex ui-components which are iteratively composed of elementary components.

Besides a formal model, the thesis motivates aspects related to an according design approach. In particular, we identified a close relationship to existing design approaches for information services. Although, our original intention was rather to provide a formal basis according to interaction patterns, the approach potentially applies to the design of information services in general.

6.2 Open Problems and Future Directions

The introduced approach and component approaches in general represent a wide research field. Therefore, in the thesis, we focused on some essential aspects, but had to neglect others. In the following, we present a brief discussion about related issues which have not yet been studied intensively. They are considered as open problems.

Context

Context plays an important role at user interaction. McCarthy et. al. [McC93, MB], for example, introduce the notion of *discourse context*. It describes the knowledge of users at particular dialog situations. Thereby, user scenarios are represented by alternating sequences of user utterances and (accordingly adapted) discourse contexts. Thereby, utterances commonly enrich the discourse context. They basically represent contexts by logical formulas which particularly permits reasoning on contexts. At the thesis, context is represented by two concepts. The static aspect of context is represented by a set of pairs of a context identifier (e.g., a context name) and an associated value of any data type. The dynamic aspect of context is represented by rules. Although, it is not obvious whether this representation is less expressive than the above, it seems to be not as adequate according to the representation of discourse context (or knowledge). Therefore, it would be beneficial to generalize the approach, such that more adequate context representations can be employed instead. Besides above alternative, there exist several approaches to formal context representation (cf., for example, [BBM95, Guh94, TACS98, KST03, Sow00, Ben98, Tho98]).

Formal verification

The formal component model based on interaction nets provides opportunities of verification. This applies for the net perspective as well as for the component perspective. Thereby, white-box verification concerns analysis of net dynamics. However, according to composition and assertion verification, black-box analysis should be usually applied. The reasons are twofold. Firstly, white-box and glass-box views might not always be provided, in particular, in the case of mixed composite specifications, i.e., specifications that contain sub-components not specified by interaction nets. Secondly, the effort of net analysis should preferably be applied once, without iteratively re-analyzing compositions. According to a black-box view, net analysis has to provide dynamic properties of the external component behavior — for example, in terms of logical formulas on streams. Thereby, black-box verification at the component perspective corresponds to the implication problem of logical formulas. However, although the proposed component model potentially permits verification, it has not yet been studied which particular classes of properties can be verified in general an which properties can be verified efficiently.

Design method

Although, the thesis indicates solutions to selected design related aspects, there are many open questions. Commonly, these questions are concerned with the changed perspective imposed by a component-based design. For example, at a component perspective, a pre-design phase should rather provide indications which components should be deployed — which contrasts issues of classical information system's design (cf. [KM98, FVFT01, Tha00, CF98]). Therefore, future work particularly addresses requirements and design methods according to design activities as component identification, adaptation, composition, and integration. However, this area currently experiences intensive research (cf., for example, [Szy02, LS00, Fra99]). Thus, results achieved in the future are expected to apply to our approach as well.

Component provision

Besides reuse of components, there are issues related with the design of the components themselves. They comprise questions as, for example, who is providing and maintaining components, or do there exist opportunities which motivate a partial integration of activities of component design and actual reuse (cf., for example, [Jaa02]).

We will conclude the thesis by reconsidering visions of software design by components. Final goals of component approaches are largely agreed upon and have been inspired decades ago (cf., for example, [McI68]). Basically, activities of the classical software design process which rather focus on new development are replaced by a design activities which focus on reuse. To achieve this goal, the notion of composition and, in particular, the notion of "intention of composition" plays an essential role. Without an opportunity of guaranteeing properties, component composition loses much of its attraction, since compositions may exhibit unexpected and undesired behavior. Therefore, a formal approach can significantly contribute to component-based design. Besides other aspects, it provides a formal notion of "intention" which permits to derive statements about the behavior of compositions based on elementary component behavior.

As we primarily focused on user interaction, we expect that component-based formalisms positively affect interface quality. In particular, an improvement according to usability of the user interfaces can be achieved. User interfaces may exhibit less restricted and thus richer opportunities of interaction — possibly approximating (rather) non-restricted human interaction. Therefore, component approaches are expected to significantly improve the current situation at which either (i) interaction is provided along rather sequential scenarios only — which strongly restricts flexibility in the case of unexpected situations, or (ii) design of flexible and adaptive interaction structures is extremely expensive according to the design process and consistency verification.

Appendix A

Detailed Formulation of Selected Interaction Patterns

The following sections present detailed descriptions of interaction patterns as they are proposed in [HDP02, Tid98]. Particularly, the presented formulations of patterns [Selectable Search Space] and [Set-Based Navigation] were introduced in collection [HDP02], and the presented formulation of pattern [Interaction History] was introduced in collection [Tid98]. Depending on the applied pattern formalism, they slightly differ in their underlying structure. Note that patterns proposed in [HDP02] primarily focus on web interfaces. However, the selected pattern formulations apply to other user interfaces as well.

A.1 Selectable Search Space

Problem statement and examples

Specify a category within which the search should be made or restricted to.

Many times users try accessing information through navigation. Nevertheless, as the information spaces become bigger, users are not always successful searching for the desired information. Therefore, having reached a certain point through navigation, they perform a search to find the desired information.

Every search activity pursues the goal of providing search results with high precision and high recall. It is also important to note that, with most search engines, it is possible to trade off precision against recall. If we increase the number of the documents retrieved, it is possible that the number of relevant documents retrieved also rises. At the same time, however, it is also likely that the number of irrelevant documents retrieved is increased, thus decreasing precision. At this point, the task of tuning a search can be greatly simplified with an analysis about the information being published and how it is structured, in order to find groups of semantically related documents/nodes.

Many companies offer their products through their websites. When a user is looking for something concrete, it is useless to search in product areas different from the one the desired product belongs to. Suppose a person is interested in buying a certain book, and the selected website also sells videos and gifts; it would make sense to allow the user to select the category of products in which the search should be performed.

A more complex case appears when it would be helpful to combine search spaces or categories. Consider a website that already has its information organized into several groups. It is possible that some information relevant to a given topic may appear in more than one of these groupings. It is the case of many companies that apply a given knowledge or technology in several areas or products, and therefore there may be reports about several aspects of the topic of search covered in different regions of the websites or, even in other websites of the same company. In such a scenario, additional flexibility is needed: combine groups/categories; but since the website already has several categories, it would be impractical to consider all possible combinations of areas.

Forces

- Users want to search over large search spaces.
- Precision and recall are mutually opposing measures, whereas effective search should have high precision and recall.
- Effectiveness of search can be improved by restricting the search space to known relevant subspaces.

Solution

Provide the user with a mechanism to select which category (sub-space) the user is going to search into. There are 2 common variants to this solution, according to the desired functionality:

- Allow users select only one search category at a time. The requirement to implement this solution is very simple: it should be possible to split the information space into disjoint sub-spaces. The same piece of information should not belong to more than one group at a time.
- Allow users to combine search areas. A grouping of checkboxes is presented to the user, with all possible areas of search. The user may select any combination of them. Although this approach is more flexible, it should be used only when the previous one is not feasible, since it requires extra navigation to a search page to select the desired categories

Consequences

Advantages:

- Higher recall and precision rates. Surprisingly, a number of websites with huge amounts of information neglect to provide their users with selectable search spaces. Rather, these sites rely on a simplified input field and offer only full searches over a website, which usually perform poorly since the search space is much larger. Furthermore, it forces the user to repeatedly refine the search by providing more detailed keyword information each time, in order to retrieve the desired information.
- Fewer searches are needed to find the desired topic, thus consuming less computing resources, and increasing user satisfaction.

Disadvantages:

- Categories usually must be determined manually.
- Having too many categories impacts in its usability. Therefore, a certain balance among the granularity of the search spaces and their quantity must be achieved.

Related patterns:

[Simple Search Interface], is usually enhanced with [Selectable Search Spaces]. It is also used in combination with [Node In Context] to automatically set the searching category.

A.2 Set-Based Navigation

Problem statement and examples

Hypermedia applications usually have to manage collections (a set of cities, a writer's books, the results of a search operation, etc). Naive designers tend to follow closely the golden rules of hypermedia design and only define links between entities that are semantically related; for example they will find the relationship between a book and its author, a painter and his painting, etc. When the link target is not a simply node, they will define an index (for example all of Van Gogh's paintings), and the user will have to move from the index to a target. To move on to another node, he will have to backtrack to the index to find another target. This introduces an unneeded burden when the reader wants to explore the whole set of target nodes. Surprisingly, even well known commercial applications (such as amazon.com) require this type of annoying back-and-forth navigation.

Solution

Consider sets as first class entities in a hypermedia application. Provide intra-set navigation controls to help the user get the ”next” and ”previous” element while he is traversing the set. Combine set-based navigation with proper indexes to make exploration easier. The set-based navigation pattern shows a simple example in which we will link nodes opportunistically (because they belong to the same set). In this way, two different Van Gogh paintings will be connected by a (set) link allowing reaching them sequentially.

When the same node may appear in two different sets, we use the [Nodes in Context] pattern; this pattern shows how to decouple basic node’s contents from those related with the actual navigation context. In this way, each time a node is accessed within a particular context (set) it will be ”decorated” (in the sense of the [Decorator] pattern) with the information corresponding to that set, such as intra-set navigation links, and contents related specifically with that context. For example, when we access a Van Gogh painting in the context of Paintings on Nature, the ”next” and ”previous” anchors will have a different meaning compared with the same links in the context of all of Van Gogh’s paintings in chronological order. In addition, it is often desirable to provide additional comments related solely to a given context. Decoupling the base information from the one corresponding to a context simplifies the design, and helps us think about these navigation paths in a more structured way.

A.3 Interaction History

Problem statement

A user performs a sequence of actions with an artifact, or navigates through it. Should the artifact keep track of what the user does with it? If so, how?

Forces

- People are forgetful of tedious details; users are not likely to remember just what they have recently done with the artifact, and computers are better at it than people are.
- Users may need to know exactly what they have just done, so they can undo their work or backtrack.
- Users may want a high-level overview of what they have done, to gain understanding that they would not get just from memory.
- Audit logs are sometimes necessary, such as with legal regulatory requirements.

- Highly interactive artifacts may generate huge amounts of recordable detail.
- Describing certain actions in a human-readable way is difficult.

Examples

- The ”history” or ”visited links” feature on a Web browser
- UNIX shell’s saved command history
- Logs of exchanged email or other social exchanges

Solution

Record the sequence of interactions as a ”history”. Keep track of enough detail to make the actions repeatable, scriptable, or even undoable, if possible. Provide a comprehensible way to display the history to the user; most artifacts that implement this pattern use a textual representation, especially [Composed Command], but that’s not a requirement. (In fact, a history for [Navigable Spaces] may be better portrayed as a state diagram, showing single steps, backtracks, etc.) If the artifact is capable of saving its state, as with [Remembered State], give the user the option of saving the history from session to session.

It may not be necessary to record every single transaction. Web browsers keep track of the visited sites, which is what the user presumably wants to know; they don’t record printing, saving, or preference changes. But the user should have some control over how big the history gets. This could take the form of a number of history records to keep, or an expire time, or a decision to discard the entire history at the close of a session.

Resulting Context: Now that the artifact has a mechanism to keep track of the history, the user may expect that those actions are scriptable; consider implementing a [Scripted Action Sequence] based on those mechanisms. When found in [Navigable Spaces], it also provides raw material for [Bookmarks], if a user can pore over the history and pick out certain points of interest.

Having a history around provides users with a set of milestones that they can use with Go Back to a Safe Place – but explicitly think about whether you want to actually undo all the history between the ”present” and the point in the history that the user wants to fall back to. The answer will depend upon your specific circumstances.

Notes: Jakob Nielsen pleads for better visualization of Web browsers’ navigation histories in his November 1, 1997 Alertbox column: ”Well, we can now sort the history list so that all the pages visited on a given site are listed together, but visualization is still missing. It would be very useful to have active sitemaps that showed the user’s

movements with footprints, showed additional detail at the current focus of attention while collapsing other regions, and also showed connections to other sites with a preview of the relevant sections of these other sites."

Related Patterns

The pattern may be used together with [Navigable Spaces], [Control Panel], [WYSIWYG Editor], [Composed Command], and [Social Space].

Bibliography

[Abo90] Gregory D. Abowd. Agents: Communicating interactive processes. In *3rd Int. Conf. on Human-Computer Interaction – INTERACT'90*, pages 143–148, Cambridge, U.K., August 1990. North Holland, Amsterdam.

[AD67] J. Annett and K. D. Duncan. Task analysis and training design. *Journal of Occupational Psychology*, 41:211–221, 1967.

[AGS97] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *Data Engineering*. IEEE Computer Society Press, 1997.

[AHW95] Alexander Aiken, Joseph M. Hellerstein, and Jennifer Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20(1):3–41, 1995.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[Ale87] Heather Alexander. *Formally-based Tools and Techniques for Human-Computer Dialogues*. Ellis Horwood, London, 1987.

[AMM98] Paolo Atzeni, Giansalvatore Mecca, and Paolo Merialdo. Design and maintenance of data-intensive web sites. In *6th Int. Conf. on Extending Database Technology – EDBT'98*, LNCS 1377, Valencia, Spain, March 1998. Springer, Berlin.

[AN01] Franz Achermann and Oscar Nierstrasz. Applications = components + scripts: A tour of piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, chapter 9. Kluwer Academic Publishers, 2001.

[Bau96] Bernd Baumgarten. *Petri-Netze: Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, 1996. (German).

[BBM95] Sasa Buvač, Vanja Buvač, and Ian A. Mason. Metamathematics of contexts. In *Fundamenta Informaticae*, 23(2/3/4). IOS Press, 1995.

[BCN92] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design. An Entity-Relationship Approach*. Benjamin Cummings, Redwood City, CA, 1992.

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens A. Szyperski. What characterizes a (software) component? *Software - Concepts and Tools*, 19(1):49–56, 1998.

[BDK01] Eike Best, Raymond Devillers, and Maciej Koutny. *Petri Net Algebra*. Springer, 2001.

[BDR98] James Bailey, Guozhu Dong, and Kotagiri Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 264–273, Seattle, Washington, October 1998. ACM Press, New York.

[Ben98] Johan van Benthem. Shifting contexts and changing assertions. In *Computing Natural Language*, pages 51–65. CSLI Publications, April 1998.

[BFJ96] Hans-Jörg Bullinger, Klaus-Peter Fähnrich, and Christian Janssen. Ein Beschreibungskonzept für Dialogabläufe bei graphischen Benutzerschnittstellen. *Informatik – Forschung und Enwicklung*, 11(2):84–93, 1996.

[Bis95] Joachim Biskup. *Grundlagen von Informationssystemen*. Vieweg, Wiesbaden, 1995. (German).

[BJ02a] Mira Balaban and Steffen Jurk. Effect preservation as a means for achieving update consistency. In *5th Int. Conf. on Flexible Query Answering Systems – FQAS'02*, LNCS 2522, pages 28–43, Copenhagen, Denmark, October 2002. Springer.

[BJ02b] Mira Balaban and Steffen Jurk. Intentions of updates – characterization and preservation. In *2nd Int. Workshop on Evolution and Change in Data Management – ECDM'02*, Tampere, Finland, October 2002. Springer.

[Bor01] Jan Borchers. *A Pattern Approach to Interaction Design*. John Wiley & Sons, New York, 2001.

[BP95] Rémi Bastide and Philippe A. Palanque. A petri net based environment for the design of event-driven interfaces. In *16th Int. Conf. on Application and Theory of Petri Nets – ATPN'95*, pages 66–83, Turin, Italy, June 1995.

[BRS+00] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, and Manfred Broy. A formal model for componentware. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of component-based systems*, chapter 9. Cambridge University Press, 2000.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive System*. Springer, New York, 2001.

[Bun99] Harry Bunt. Context representation for dialog management. In *2nd Int. and Interdisciplinary Conf. on Modeling and Using Context – CONTEXT'99*, LNCS 1688, pages 77–90, Trento, Italy, September 1999. Springer.

[BW00] Elena Baralis and Jennifer Widom. An algebraic approach to static analysis of active database rules. *ACM Transactions on Database Systems*, 25(3):269–332, 2000.

[CF98] Wolfram Clauss and Thomas Feyer. Challenges in integrated information services design, October 1998. Working Group Proposal at the 7th Int. Workshop on Foundations of Models and Languages for Data and Objects – FMLDO'98.

[CFP99] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. Data-driven, one-to-one web site generation for data-intensive applications. In *25th Int. Conf. on Very Large Data Bases – VLDB'99*, pages 615–626, Edinburgh, Scotland, September 1999.

[CL99] Wolfram Clauss and Jana Lewerenz. Abstract interaction specification for information services. In *Int. Conf. On Managing Information Technology Resources in Organizations*, Hershey, Pennsylvania, May 1999.

[Cod70] Edgar F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[CPR01] Rosa M. Carro, Estrella Pulido, and Pilar Rodríguez. Improving website maintenance with TANGOW by making page structure and contents independent. In *Web Engineering, Software Engineering and Web Application Development*, LNCS 2016, pages 325–334. Springer, 2001.

[CT97] Wolfram Clauss and Bernhard Thalheim. Abstraction layered structure-process codesign. In D. Janaki Ram, editor, *Management of Data*. Narosa Publishing House, New Delhi, 1997.

[DFAB98] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction, 2nd Edition*. Prentice-Hall, 1998.

[DLH02] Douglas K. van Duyne, James A. Landay, and Jason I. Hong. *The Design of Sites: Patterns, Principles, and Processes For Crafting a Customer-Centered Web Experience*. Addison-Wesley, 2002.

[DT02] Antje Düserhöft and Bernhard Thalheim. Integrating retrieval functionality in web sites based on storyboard design and word fields. In *6th Int. Conf. on Applications of Natural Language to Information Systems – NLDB'02*, pages 52–63, Stockholm, Sweden, June 2002.

[EN99] Ramez A. Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 3rd edition, 1999.

[FFK+98] Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, and Dan Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 1998. ACM Press.

[FKST00] Thomas Feyer, Odej Kao, Klaus-Dieter Schewe, and Bernhard Thalheim. Design of data-intesive web-based information services. In *1st Int. Conf. on Web Information Systems Engineering*, pages 462–467. IEEE Computer Society Press, June 2000.

[FLM98] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the World-Wide Web: A survey. In *ACM SIGMOD International Conference on Management of Data*, volume 27(2), pages 414–425. ACM Press, June 1998.

[FP98] Piero Fraternali and Paolo Paolini. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In *6th Int. Conf. on Extending Database Technology – EDBT'98*, LNCS 1377, Valencia, Spain, March 1998.

[Fra99] Ulrich Frank. Component Ware – Software-technische Konzepte und Perspektiven für die Gestaltung betrieblicher Informationssysteme. *Information Management & Consulting*, 14(2):11–18, 1999. (German).

[FST98] Thomas Feyer, Klaus-Dieter Schewe, and Bernhard Thalheim. Conceptual design and development of information services. In *17th Int. Conf. on Conceptual Modeling – ER'98*, LNCS 1507, Singapore, November 1998. Springer, Berlin.

[FT99] Thomas Feyer and Bernhard Thalheim. E/R based scenario modeling for rapid prototyping of web information services. In *Int. Workshop on the World-Wide Web and Conceptual Modeling – WWWCM'99*, LNCS 1727. Springer, Berlin, November 1999.

[FT02a] Thomas Feyer and Bernhard Thalheim. Many-dimensional schema modeling. In *6th East-European Conf. on Advances in Databases and Information Systems – ADBIS'02*, LNCS 2435, Bratislava, Slovakia, September 2002. Springer.

[FT02b] Thomas Feyer and Bernhard Thalheim. A model for defining and composing interaction patterns. In *12th European-Japanese Conf. on Information Modelling and Knowledge Bases – EJC'02*, Krippen, Germany, May 2002. IOS Press, Amsterdam.

[FT03] Thomas Feyer and Bernhard Thalheim. Component-based interaction design. In *13th European-Japanese Conf. on Information Modelling and Knowledge Bases – EJC'03*, Kitakyushu, Japan, June 2003.

[FVFT01] Thomas Feyer, Marcela Varas, Marta Fernández, and Bernhard Thalheim. Intensional logic for integrity constraint specification in predesign database modeling. In *11th European-Japanese Conf. on Information Modelling and Knowledge Bases — EJC'01*, Maribor, Slowenia, May 2001. IOS Press, Amsterdam.

[GC99] D. M. Germán and D. D. Cowan. Formalizing the specification of web applications. In *Int. Workshop on the World-Wide Web and Conceptual Modeling – WWWCM'99*, LNCS 1727. Springer, Berlin, November 1999.

[GC00] D. M. Germán and D. D. Cowan. Towards a unified catalog of hypermedia design patterns. In *33nd Int. Conf. on System Sciences – HICSS-33*, Maui, Hawaii, January 2000.

[GG00] Martin Gaedke and Guntram Gräf. Development and evolution of Web-applications using the WebComposition process model. In *Int. Workshop on Web Engineering at the 9th Int. World-Wide Web Conf.*, Amsterdam, The Netherlands, May 2000. Springer.

[GGS+99a] Martin Gaedke, Hans-W. Gellersen, Ablrecht Schmidt, Ulf Stegemüller, and Wofgang Kurr. Hypermedia patterns and components for building better Web information systems. In *Int. Workshop on Hypermedia Development – Design Patterns in Hypermedia – Hypertext'99*, Darmstadt, Germany, February 1999.

[GGS+99b] Martin Gaedke, Hans-W. Gellersen, Ablrecht Schmidt, Ulf Stegemüller, and Wofgang Kurr. Object-oriented web engineering for large-scale web service management. In *32nd Int. Conf. on System Sciences – HICSS-32*, Wailea, Hawaii, January 1999.

[GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *7th European Conf. on Object-Oriented Programming – ECOOP'93*, LNCS 707, Kaiserslautern, Germany, July 1993. Springer.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GL97] Marc Gyssens and Laks V. S. Lakshmanan. A foundation for multi-dimensional databases. In *23rd Int. Conf. on Very Large Data Bases – VLDB'97*, Athens, Greece, August 1997.

[GPBV99] Franca Garzotto, Paolo Paolini, Davide Bolchini, and Sara Valenti. "Modeling-by-patterns" of Web applications. In *Int. Workshop on the World-Wide Web and Conceptual Modeling – WWWCM'99*, LNCS 1727, Paris, France, November 1999. Springer, Berlin.

[Guh94] Ramanathan V. Guha. *Contexts: A Formalization and some Applications*. PhD thesis, Stanford University, 1994.

[GV02] Claude Girault and Rüdiger Valk. *Petri Nets for System Engineering. A Guide to Modeling, Verification, and Applications*. Springer, 2002.

[HDP02] Hypermedia design patterns repository. ACM-SIGWEB in collaboration with the University of Italian Switzerland, Online version: http://www.designpattern.lu.unisi.ch/index.htm, 2002.

[Hei01] Birk Heinze. *Entwurf adaptiver Informationsdienste unter Benutzung von Interaktionspattern*. Master thesis, Brandenburg University of Technology at Cottbus, Germany, 2001. (German).

[HF99] Birk Heinze and Thomas Feyer. Entwurf von Informationsdiensten für das Web: Modellierung von Benutzerszenarien auf Grundlage des multidimensionalen Datenmodells. In *11. Workshop Grundlagen von Datenbanken, Jeaner Schriften zur Mathematik und Informatik*, Math/Inf/99/16, Luisenthal, Germany, May 1999.

[HLP97] Martin G. Helander, Thomas K. Landauer, and Prasad V. Prabhu, editors. *Handbook of Human-Computer Interaction, 2nd Edition*. North Holland, Amsterdam, 1997.

[HP98] David Harel and Michal Politi, editors. *Modeling reactive systems with statecharts — the STATEMATE approach*. McGraw-Hill, New York, 1998.

[Jaa02] Hannu Jaakkola. The extensive role of reuse in software engineering. In *Slovenian Informatics Conference – DSI'02*, pages 1–16, Portorož, Slowenia, April 2002.

[Jen97a] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 2. Springer, 1997.

[Jen97b] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer, 1997.

[Jen02] Kurt Jensen et. al. Design/CPN. A reference manual. Online version: http://www.daimi.aau.dk/designCPN/, Meta Software and Computer Science Department, University of Aarhus, Denmark, 2002.

[KM98] Christian Kop and Heinrich C. Mayr. Conceptual predesign — bridging the gap between requirements and conceptual design. In *3rd Int. Conf. on Requirements Engineering – ICRE'98*, pages 90–100, Colorado Springs, Colorado, April 1998.

[KST03] Roland Kaschek, Klaus-Dieter Schewe, and Bernhard Thalheim. Modelling contexts in web information systems. In *15th Int. Conf. on Advanced Information Systems Engineering – CAiSE'03*, LNCS 2681, Klagenfurt, Austria, June 2003.

[LAN00] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A formal language for composition. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of component-based systems*, chapter 4. Cambridge University Press, 2000.

[Lew00] Jana Lewerenz. *Human-Computer Interaction in Heterogeneous and Dynamic Environments: A Framework for its Conceptual and Automatic Customisation*. PhD thesis, Brandenburg University of Technology at Cottbus, Germany, 2000.

[LS00] Gary T. Leavens and Murali Sitaraman, editors. *Foundations of component-based systems*. Cambridge University Press, 2000.

[LST99] Jana Lewerenz, Klaus-Dieter Schewe, and Bernhard Thalheim. Modelling data warehouses and OLAP applications by means of dialog objects. In *18th Int. Conf. on Conceptual Modeling – ER'99*, LNCS 1728. Springer, Berlin, November 1999.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. In *ACM SIGMOD International Conference on Management of Data*, volume 26(3), pages 54–66, Tucson, Arizona, May 1997. ACM Press.

[MAM03] Paolo Merialdo, Paolo Atzeni, and Giansalvatore Mecca. Design and development of data-intensive web sites: The araneus approach. *ACM Transactions on Internet Technology*, 3(1):49–92, 2003.

[MB] John McCarthy and Sasa Buvač. Formalizing context (expanded notes). *http://www-formal.stanford.edu/jmc/mccarthy-buvac-98/index.html*.

[McC93] John McCarthy. Notes on formalizing context. In *Int. Joint Conf. on Artificial Intelligence*, Chambéry, France, January 1993.

[McI68] M. Douglas McIllroy. Mass produced software components. In *NATO Conference on Software Engineering*, pages 138–155, Garmisch, Germany, October 1968. Scientific Affairs Division, NATO.

[New68] William M. Newman. A system for interactive graphical programming. In *Spring Joint Computer Conf. – AFIPS*, pages 47–54, Atlantic City, New Jersey, April 1968. Thomson Book Company, Washington D.C.

[Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1994.

[Nie00] Jakob Nielsen. *Designing Web Usability*. Indianapolis, New Riders, 2000.

[NL95] William M. Newman and Michael G. Lamming. *Interactive System Design*. Addison-Wesley, 1995.

[NL00] Mark W. Newman and James A. Landay. Sitemaps, storyboards, and specifications: A sketch of web site design practice. In *Int. Conf. on Designing Interactive Systems: Processes, Practices, Methods, Techniques — DIS'00*, pages 263–274. ACM Press, New York, August 2000.

[Par69] David L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *24th National ACM Conference*, pages 379–385, 1969.

[Rad01] Michael Radigk. *Automatische Generierung graphischer Userinterfaces unter Beachtung der Benutzer- und Gerätemodelle durch Entwicklung einer Treiberhierarchie*. Master thesis, Brandenburg University of Technology at Cottbus, Germany, 2001. (German).

[Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer, 1998.

[RSL99] Gustavo Rossi, Daniel Schwabe, and Fernando Lyardet. Web application models are more than conceptual models. In *Int. Workshop on the World-Wide Web and Conceptual Modeling – WWWCM'99*, LNCS 1727. Springer, Berlin, November 1999.

[RSL00] Gustavo Rossi, Daniel Schwabe, and Fernando Lyardet. User interface patterns for hypermedia application. In *Int. Working Conf. on Advanced Visual Interfaces – AVI'00*, pages 136–142, Palermo, Italy, May 2000. ACM Press, New York.

[See98] Kati Seelig. *Zielplattformunabhängige Konzepte für die Dialogverwaltung in einem integrativen Entwurfsmodell*. Student research report, Brandenburg University of Technology at Cottbus, Germany, 1998. (German).

[SF02] Ahmed Seffah and Peter Forbrig. Multiple user interfaces: Towards a task-driven and patterns-oriented design model. In *9th Int. Workshop on Interactive Systems. Design, Specification, and Verification – DSV-IS 2002*, LNCS 2545, pages 118–132, Rostock, Germany, June 2002. Springer.

[SN99] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures – Advances and Applications*, chapter 2, pages 13–25. Springer, 1999.

[Sow00] John F. Sowa. *Knowledge Representation: Logical, philosophical, and computational foundations*. Brooks/Cole, 2000.

[SR98] Daniel Schwabe and Gustavo Rossi. Developing hypermedia applications using OOHDM. In *Workshop on Hypermedia Development*, Pittsburgh, Pennsylvania, June 1998.

[SREL00] Daniel Schwabe, Gustavo Rossi, Luiselena Esmeraldo, and Fernando Lyardet. Web design frameworks: An approach to improve reuse in web applications. In *Int. Workshop on Web Engineering at the 9th Int. World-Wide Web Conf.*, Amsterdam, The Netherlands, May 2000. Springer.

[Sri01] Srinath Srinivasa. *An Algebra of Fixpoints for Characterizing Interactive Behavior of Information Systems*. PhD thesis, Brandenburg University of Technology at Cottbus, Germany, 2001.

[ST98] Klaus-Dieter Schewe and Bernhard Thalheim. Limitations of rule triggering systems for integrity maintenance in the context of transition specifications. *Acta Cybernetica*, 13(3):277–304, 1998.

[ST99] Klaus-Dieter Schewe and Bernhard Thalheim. Towards a theory of consistency enforcement. *Acta Informatica*, 36(2):97–141, 1999.

[Szy98] Clemens A. Szyperski. Emerging component software technologies – a strategic comparison. *Software - Concepts and Tools*, 19(1):2–10, 1998.

[Szy02] Clemens A. Szyperski. *Component Software. Beyond object-oriented programming*. Addison-Wesley, 2nd edition, 2002.

[TACS98] Manos Theodorakis, Anastasia Analyti, Panos Constantopoulos, and Nicolas Spyratos. Context in information bases. In *3rd Int. Conf. on Cooperative Information Systems – CoopIS'98*, New York City, New York, August 1998. IEEE Computer Society Press.

[TBF+98] Bernhard Thalheim, Cornell Binder, Thomas Feyer, Thomas Gutacker, and Srinath Srinivasa. Konzeptioneller Entwurf und Gestaltung von internet- und kabelnetzbasierten Bürgerinformationsdiensten. In *Netzinfrastrukturen und Anwendugen für die Informationsgesellschaft – INFO'98*, Potsdam, Germany, October 1998. (German).

[TD03] Bernhard Thalheim and Antje Düsterhöft. Systematic development of internet site: Extending approaches of conceptual modeling. In Patrick van Bommel, editor, *Information Modeling for Internet Applications*, chapter 5, pages 80–102. Idea Group Publications, 2003.

[Tha93] Bernhard Thalheim. Fundamentals of entity-relationship modelling. *Annals of Mathematics and Artificial Intelligence*, 7(1–4):197–256, 1993.

[Tha00] Bernhard Thalheim. *Entity-Relationship Modeling – Foundations of Database Technology*. Springer, Heidelberg, 2000.

[Tho98] Richmond H. Thomason. Representing and reasoning with context. In *Int. Conf. on Artificial Intelligence and Symbolic Computation – AISC'98*, LNCS 1476, pages 29–41, New York City, New York, September 1998. Springer.

[Tid98] Jenifer Tidwell. Interaction design patterns. In *Int. Conf. on Pattern Languages of Programming – PLoP'98*, Monticello, Illinois, 1998. extended version at www.mit.edu/~jtidwell/interaction_patterns.html.

[VLF00] Vojtech Vestenicky, Jana Lewerenz, and Thomas Feyer. Modelling the modification component of an information service. In *4th East-European Conference on Advances in Databases and Information Systems – ADBIS'00, Proceedings of Chalenges*, pages 195–204, Prague, Czech Republic, September 2000. Matfyz Press.

[Was85] Anthony I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, 11(8):699–713, 1985.

[WT00a] Martin van Welie and Hallvard Trætteberg. Interaction patterns in user interfaces. In *Pattern Languages in Program Design – PLoP'00*, Monticello, Illinois, August 2000.

[WT00b] Martin van Welie and Hallvard Trætteberg. Patterns as tools for user interface design. In *Int. Workshop on Tools for Working with Guidelines*, pages 313–324, Biarritz, France, October 2000.