

# Model-based User Interface Design

**Hallvard Trætteberg**  
**Email: [hal@idi.ntnu.no](mailto:hal@idi.ntnu.no)**

**Information Systems Group**  
**Department of Computer and Information Sciences**  
**Faculty of Information Technology, Mathematics and Electrical Engineering**  
**Norwegian University of Science and Technology**

**14 May 2002**

## Abstract

This work is about supporting user interface design by means of explicit design representations, in particular models.

We take as a starting point two different development traditions: the formal, analytic, top-down engineering approach and the informal, synthetic, bottom-up designer approach. Both are based on specific design representations tailored to the respective approaches, and are found to have strengths and weaknesses. We conclude that different representations should be used during user interface design, based on their specific qualities and the needs of the design process.

To better understand the use of design representations a framework for classifying them is developed. A design representation may be classified along three dimensions: the perspective (problem- or solution-oriented) of the representation, the granularity of the objects described and the degree of formality of the representation and its language. Any design approach must provide representation languages that cover the whole classification space to be considered complete. In addition, the transitions between different representations within the representation space must be supported, like moving between task-based and interaction-oriented representations or up and down a hierarchic model. Movements between representations with different degrees of formality are particularly important when combining user-centered design with a model-based approach.

The design representation classification framework has guided the development of diagram-based modelling languages for the three main perspectives of user interface design, tasks, abstract dialogue and concrete interaction. The framework has also been used for evaluating the languages. A set-based conceptual modelling language is used for domain modelling within all these perspectives. The task modelling language is designed as a hybrid of flow-oriented process languages and traditional hierarchical sequence-oriented task languages. Key features are tight integration with the domain modelling language, expressive and flexible notation and support for classification of task structures. The language for modelling abstract dialogue is based on the interactor abstraction for expressing composition and information flow, and the Statecharts language for activation and sequencing. Parameterized interactors are supported, to provide means of expressing generic and reusable dialogue structures. Modelling of concrete interaction is supported by a combination of the dialogue and domain modelling languages, where the former captures the functionality and behavior and the latter covers concepts that are specific for the chosen interaction style.

The use of the languages in design is demonstrated in a case study, where models for tasks, dialogue and concrete interaction are developed. The case study shows that the languages support movements along the perspective, granularity and formality dimensions.



## Preface

This is a doctoral thesis submitted for the degree “doktor ingeniør” to the Norwegian University of Science and Technology (NTNU). The work has been supervised by Professor Arne Sølvberg at the Information Systems group, Dept. of Computer and Information Sciences, Faculty of Information Technology, Mathematics and Electrical Engineering at NTNU. The Research Council of Norway (NFR) has funded the work under Grant No. 116388/410.

My interest in abstract models for user interface design was initiated by practical experiences with Lisp-based development environments at Center for Industrial Research (SI) in Oslo. Arne Sølvberg was kind to let my work on this topic on my diploma thesis in autumn -90 and spring -91, where I developed abstract dialogue modelling concepts very similar to those presented later in this thesis. During my subsequent five years at SI (later part of SINTEF), I got more practical experience with user interface design tools and continued toying with ideas and concepts for *model-based* user interface design. In autumn '96 I received funding for a dr. project by NFR, and Arne Sølvberg again accepted to supervise me.

My starting point was abstract dialogue modelling and the initial work was on models for interaction. However, the general interest in task-based design and the work of Arne's student Steinar Carlsen on workflow modelling, prompted me to look at task analysis and modelling. This was an important transition, as it moved the focus from the user interface to the user and his needs, goals and practice. Dag Svanæs then introduced me for user-centered and iterative design, convinced me of the need for communicating with end-users by means of concrete design representations and made me aware of different design traditions. This gave rise to the framework for the thesis as a whole.

I want to thank my supervisor, Professor Arne Sølvberg, for his continuous support of my work, both theoretical, practical and personal. Arne Sølvberg and his group have provided a stimulating environment for me, for which fellow students Terje Brasethvik and Babak Farshchian deserve special thanks. Steinar Carlsen has been an important colleague both as my boss at SI/SINTEF where my initial inspiration was fostered, and as a fellow student paving the way for part of my work. I am very grateful for his wisdom and humor.

I also thank Roald Kvamstad and Anita Kvamme, both part of the Gazelle team at Statoil, who provided me with a realistic case for trying out my modelling languages.

Last but not least, I want to thank my wife Marit Reitan, for her love and support, and much needed understanding when the project was delayed. My kids Jens and Anne, both born during this work, also deserve hugs and kisses for providing me with alternative fun and stimuli and helping me keep in touch with the real world.



# Table of Content

<i>Chapter 1 Introduction</i> .....	<i>1</i>
1.1 Problem statement and objectives .....	2
1.2 Approach .....	3
1.3 Contributions .....	4
1.4 Outline of the thesis .....	5
<i>Chapter 2 State of the Art</i> .....	<i>9</i>
2.1 User interface modelling .....	9
2.2 Obstacles to formal interface design and development .....	13
2.3 Informal vs. formal representations .....	15
2.4 User interface design within a systems development context .....	16
2.4.1 UML and interaction design .....	17
2.5 Summary .....	20
<i>Chapter 3 Design and representation</i> .....	<i>21</i>
3.1 High-level view of People, Actions, Information and Tools .....	21
3.2 Engineering and designer traditions .....	24
3.3 Three dimensions of design representations .....	26
3.4 A fourth dimension: user - system? .....	31
3.5 The role of design representations .....	33
3.6 Representations and process .....	35
3.7 Formal representations revisited .....	39
<i>Chapter 4 Task modelling</i> .....	<i>41</i>
4.1 Introduction .....	41
4.2 Workflow and task modelling .....	42
4.3 Aspects of work and its description .....	43
4.4 Workflow and task models .....	49
4.5 Towards a task modelling language .....	50
4.6 The domain modelling language: RML .....	52
4.6.1 Concepts and instances .....	52
4.6.2 Relations between concepts .....	54
4.6.3 Dynamics .....	58
4.7 The Task Modelling Language - TaskMODL .....	59
4.7.1 Tasks and subtasks .....	61
4.7.2 Resources .....	63
4.7.3 Subtask sequence constraints .....	66
4.7.4 Subtask cardinality .....	68
4.7.5 Explicit pre-conditions and post-conditions .....	69
4.7.6 The task life-cycle, resource binding and data flow .....	71
4.7.7 Task classification .....	73
4.7.8 Generalisation and specialisation .....	75

4.8 Conclusion .....	79
4.8.1 RML and TaskMODL and the representation framework.....	80

## Chapter 5 Dialogue modelling ..... 83

5.1 Introduction .....	83
5.2 Interactors as information mediators .....	84
5.2.1 Gates, connections and functions.....	86
5.2.2 Basic device interactors.....	91
5.2.3 Set-oriented device interactors .....	93
5.3 Interactor content.....	95
5.3.1 Interactor control.....	95
5.3.2 Interactor composition.....	98
5.4 Modelling system functionality .....	100
5.5 Interactor resources and parameters .....	101
5.6 Acts and functions .....	105
5.6.1 Acts.....	106
5.6.2 Act classification.....	108
5.6.3 Act invocation .....	109
5.6.4 Function and gate decomposition.....	111
5.7 Conclusion.....	114
5.7.1 DiaMODL and the representation framework.....	115

## Chapter 6 Concrete interaction ..... 119

6.1 Introduction .....	119
6.2 Window-based interaction objects.....	120
6.3 Simple interaction objects .....	123
6.3.1 Labels and icons.....	123
6.3.2 Buttons .....	124
6.3.3 Popups dialogs, dropdown listboxes and menus.....	126
6.4 Composite interaction objects .....	129
6.4.1 Composition of function.....	129
6.4.2 Composition of control.....	132
6.4.3 Composition of structure.....	134
6.5 Direct manipulation and mouse gestures.....	135
6.5.1 Structure of recognisers .....	137
6.6 Composition of gestures .....	142
6.7 Conclusion.....	144
6.7.1 Concrete interaction and the representation framework.....	145

## Chapter 7 Case study..... 147

7.1 The IFIP case .....	147
7.2 The ‘Organize’ task .....	148
7.3 The domain model.....	152
7.4 The main subtasks .....	153
7.4.1 Task A.2, Record response.....	153
7.4.2 Task A.3, Choose reviewers.....	154
7.4.3 Task A.4, Perform review .....	155
7.4.4 Task A.5, Collect review results .....	155
7.5 Dialogue-oriented domain model - considerations.....	156
7.6 Designing a dialogue for each main subtask .....	157
7.6.1 Registering a paper.....	158
7.6.2 Assigning a set of reviewers to a paper .....	159

7.6.3 Collecting and summarizing review results .....	160
7.7 Composing the dialogues - considerations .....	162
7.8 From abstract dialogue to concrete interaction.....	163
7.9 Completing the design.....	166
7.10 Conclusion .....	168

## *Chapter 8 Experiences and feedback ..... 169*

8.1 External presentations and teaching .....	169
8.2 The Statoil case.....	170
8.3 Tools and implementations .....	174
8.3.1 Designing the notation and drawing diagrams with Visio.....	174
8.3.2 Garnet and CLIM-based GUI-builder prototypes .....	175
8.3.3 Java-implementation of Statecharts for direct manipulation.....	177
8.4 Concluding remarks.....	177

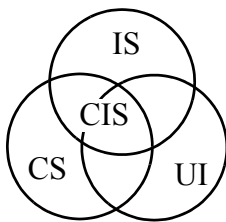
## *Chapter 9 Conclusion and Future work ..... 179*

9.1 Main contributions.....	180
9.2 Limitations and Future work .....	181
9.2.1 Model-based design patterns .....	182
9.2.2 UML integration .....	183
9.2.3 Building User Interfaces with interactors.....	187
9.2.4 Summary.....	190



# Chapter 1

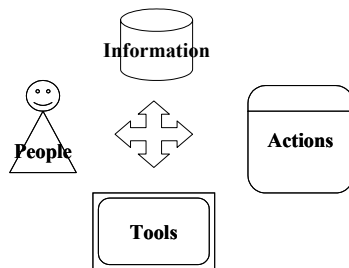
## Introduction



**Figure 1.** Information System (IS), User Interface (UI) and Computer System (CS)

This work is about integrated methods for model-based development of user interfaces, in the context of information systems development. *Information systems* (IS) are compositions of information processing entities, working together in the pursuit of goals. Most organisations can be considered an information system in this sense, since reaching their *business goals* normally requires decision making and information processing. A *Computerised Information Systems* (CIS) is the part of a larger IS that is implemented by a *Computing System* (CS), built as a tool for aiding the human or organisational part of an IS. Figure 1 illustrates that the CIS is the overlapping part of an Information System and a Computer System.

The *users* of a CIS interact with it through its *user interface* (UI) in the pursuit of organisational *goals*. By letting the UI in Figure 1 *partly* overlap the IS, CIS and CS, we indicate that the user in the pursuit of personal goals, may interact with other parts of the CS, as well as non-computerised artifacts that may or may not be part of the IS. In all cases, people use tools to perform the actions judged necessary for reaching their goals, as illustrated in Figure 2. However, the context and scope of the goals and actions are of course different.



**Figure 2.** People use information and tools to perform actions

Information systems *construction* and user interface *design* have traditionally been undertaken using different methods and techniques, and the people taking part in these activities have often been part of different academic traditions. Such a separation is unfortunate, since every information system has a user interface which for many purposes is perceived as the whole system. For the user, the distinction between interface and system is, for most practical purposes, meaningless. In addition, most application usage is part of a larger information processing context, whether computerised or not. Thus, isolating the design of the application's user interface from the information system development is clearly undesirable. However, the conceptual distinction between user interface and information system remains useful and interesting. The proliferation of different connected electronic devices necessitates the development of many front-ends for the same information system. The different capabilities of these devices result in differing interfaces, while the information system remains essentially the same. Furthermore, many future products may serve as new inter-

faces for existing and largely unchanged systems. Hence, speaking about the information system and its separately attached user interfaces is clearly meaningful.

The concept of an *interactive system* seems to appreciate the duality and the need for integration between systems development and interface design. Managing this integration as well as their separate concerns is key to successful development of interactive systems.

## 1.1 Problem statement and objectives

Within the tradition of information systems engineering and building on the work within model-based user interface design methodology, this work addresses the following problem:

*How can models be used as a representation tool in the development of interactive systems and provide a means of integrating information systems development with user interface design?*

The established models of information systems engineering have many important properties, where formality has perhaps been the most appreciated one. However, the underlying conceptual abstractions of entities and properties, events and transformations, are perhaps more important than their formalisation, because they support mental synthesis and analysis within a domain. In addition, the use of graphical representations or diagrams (“a drawing that shows arrangement and relations (as of parts)”) for depicting relations and composition, is seen as important, as it enhances the understanding of the complex structures that are needed to solve complex problems. Hence, basing our work on models from the field of information systems engineering seems like a sound approach.

On the other hand, current methods and techniques for user interface development within more design(er)-oriented traditions, show that there is a need for flexible representations that stimulate creativity and enhance communication. By flexible we mean the ability to combine different aspects of the domain in one representation, without constraining the allowed content or form.

This leads up to the following aims of this work:

1. To provide a framework for understanding the use of user interface design representations in general, and more formal models in particular.
2. To design an integrated family of flexible diagram-based languages for specifying/describing both the problem domain and the interface solution. The languages should be interoperable with or based on languages used in information systems engineering.
3. To outline how models using these languages can be used for capturing design knowledge for reuse across problems and projects.

## 1.2 Approach

The main topics of this work are user interface modelling and model-based design of user interfaces, which both are established fields. Two identified needs have motivated this work, both having to do with integration of models and methods:

- The integration of model-based design of user interfaces with information systems development methods.
- The integration of the various modelling languages used within model-based design of user interfaces.

A third concern is based on the differences between information systems engineering and user interface design traditions, and provides an important constraint:

- Making the user interface models more accessible to developers untrained in formal modelling.

All these points can be discussed along two dimensions, process and representation. The process dimension is concerned with how organisations choose to structure the development, what activities are included, their timing and dependencies, who participate and which practical techniques that are used. The representation issue is concerned with how knowledge, problems and solutions are articulated and expressed, and the specific content and form of these explicit expressions. Process and representation is interrelated, as certain activities rely on a certain kind of representation, and certain representations are fit for certain processes. We choose to approach the integration problem along the representation dimension and focus on the development of modelling languages. We do not wish to endorse a particular process, as this is more of an organisational issue.

Being agnostic about the development process still means that the role of representations in the process must be taken into account. We do this by first characterising the engineering and designer traditions and identifying their strengths and weaknesses. This is used as the basis for suggesting a framework for classifying representations, with dimensions for content and form. Furthermore, we identify processes necessary during development for use and transformation of representations.

Integration of models and methods must happen along both the content and form dimensions. Along the content dimension, we approach integration by taking representative languages from the information systems engineering tradition as a starting point for our user interface modelling languages:

1. Our task modelling language TaskMODL is based in the APM workflow language [Carlsen, 1998], whose main concepts are interpreted in the context and tradition of task analysis and modelling. The traditional simplicity and hierarchical tree style of existing task languages is in TaskMODL integrated with APM's containment style, to better support both traditions.

2. The Referent domain modelling language [Sølvberg, 1999] developed by our group, is integrated with both TaskMODL and the proposed dialogue modelling language (DiaMODL), acting as a glue both between system and interface, and between different interface perspectives.
3. The dynamic part of DiaMODL, is a hybrid of the classical interactor-with-gates interface component abstraction and the hierarchical state formalism Statecharts.

Along the form dimension, we approach the problem of integration in two ways:

1. We try to design flexible languages, with support for stepwise refinement in a meaningful way. This should make the use of these languages less constraining.
2. We provide a direct mapping from abstract DiaMODL constructs to concrete design-oriented elements, and suggest a way of interchanging abstract and concrete elements to make the dialogue models more understandable and mentally stimulating.

The integrated use of our proposed languages is tested on the same case study used for validating APM. This prompts a discussion of which knowledge is needed for doing design. We propose a design pattern approach to the management of design knowledge. Design patterns relate model fragments in our languages, and provide guidance for how to move within the design space.

Our focus is on supporting the expression of statements about the problem and the elements and structure of its proposed solution. Therefore we downplay the formal aspects of our languages. Still, prototype implementations of the dialogue modelling language, are used to validate that our approach can be used for interface prototyping and implementation purposes.

## 1.3 Contributions

The major contributions of this work are:

- We identify different roles of design representations during interface development and proposed a framework for classifying design representations. Both contribute to an enhanced understanding of when and how different design representations should be used in the development process.
- We propose a task modelling language that comprises:
  - *An advanced domain modelling language*, which is used for modelling the static part of the problem domain. The language is simple and expressive, and integrated both conceptually and visually with the dynamic part.
  - *Comprehensive support for modelling resources*, such as performing actor, required information and tools, which gives uniform treatment of information and resource flow and parameterization.

- *Support for generalisation/specialisation*, providing support for capturing general and specific task structures.
- *A flexible visual syntax* supporting a hybrid of hierarchical tree structure and surface containment, and integrating the static and dynamic aspects of the domain.
- An interactor-based dialogue modelling language has been proposed, featuring:
  - *Generic functional abstraction* with scalable support for composition.
  - *Simplified activation semantics* through integration with Statecharts [Harel, 1987].
  - Integration with task modelling, through the use of dynamic constructs similar to the task modelling language and the same domain modelling language.
  - *Straight forward concrete interpretation* in terms of concrete dialogue elements, supporting a smooth transition to concrete design
  - *Parameterized interactors*, supporting generic dialogue structures
- Models of concrete interaction, based on the dialogue modelling language:
  - elements of standard *window-based* interfaces and their composition
  - the *direct manipulation* style of interaction

## 1.4 Outline of the thesis

This work has the goal of building bridges and crossing boundaries between different design representations and traditions. There are two major boundaries that are targeted. The first concerns the use of informal and formal representations, which have different form and focus and differing values attached to them. Building a bridge between these two will make it easier to support different modes of work and enable practitioners within different traditions to work better together. The second boundary is found between representations with different focus and content, which exists in the formal tradition. Building bridges between these, will make it easier to support iterative processes, with a mix of prototyping, forward and backward engineering and top-down and bottom-up design.

In Chapter 2, “State of the Art”, a framework for classifying design representations is presented. The framework defines a design representation space with three axes, within which different design representation are placed. It is suggested that the process of design requires moving within this space, i.e. using design representations with different characteristics, content and focus. Several design representations are needed, both to move from high-level goals and problem statements to suggestions for solutions, but also to support different modes of design, e.g. synthesis and analysis, creativity and critique. The need for using a particular design representation depends on its characteristics and the role we want it to play within the development process, i.e. the requirements we have to the design representation, which may vary considerably during the development of a software product. The chapter will conclude by reviewing work within different areas covered by the design representation space.

In the chapters 4, 5 and 6, we focus on three particular perspectives of user interface design. Throughout these chapters we will present many models with named model elements. When using these names as (implicit) references to the model elements, we will use a special CHARACTER FONT, to distinguish them from other words. In Chapter 4, “Task modelling”, we discuss the problem-oriented perspective, combining static modelling of the domain with models of human action. The goal is to provide a modelling language with a clear focus, so the designer naturally expresses the problem and not its solution. In addition, a flexible notation is important, to avoid hindering the designer in the process of expressing constraints on action sequences. First, a domain language is presented, based on a conceptual modelling language previously used for meta-modelling, problem analysis and requirements engineering. Second, a task modelling language is presented, based on a combination of a dataflow-oriented workflow language and more traditional hierarchical task analysis model. The static domain language is used both together with the dynamic task language to model the task domain, and for defining the meaning of the task modelling constructs. However, the goal in the latter case is not to give a fully formal definition of the task language, but rather to explain the notation in terms of the underlying concepts of the task domain.

In Chapter 5, “Dialogue modelling”, the topic will be the solution-oriented perspective of the dialogue between human and interactive system, i.e. the abstract design of how human and system cooperate in performing the desired tasks. A modelling language based on the interactor abstraction for information mediation is presented. The goal of the language is to simplify stepwise refinement of design and ease the transition to and from more concrete representations. The resulting language has weaker compositional characteristics than previous interactor definition, but should be easier to use and provide more suitable means of abstraction.

In Chapter 6, “Concrete interaction”, we focus on the concrete building blocks which are used when composing user interfaces. Models of the dialogue elements which are part of some concrete interaction styles are presented, and we take a more detailed look the correspondence with the previously presented interactor model. Based on the interactor model and how it provides abstract elements from which dialogues are composed, we discuss the constraints for composing concrete elements within particular interaction styles.

In Chapter 7, “Case study”, we present a design case and show how the modelling languages presented in chapters 4 and 5 may be used for modelling and design. The goal is to show how the languages support moving forward along the perspective axis of the design representation classification framework, i.e. moving from problem to solution. We first develop a domain and task model and show how they may inform the designer and guide the development of the abstract dialogue model. We then choose an interface style and proceed by developing a concrete design, again guided by the more abstract model.

As a further validation of our approach, we in Chapter 8, “Experiences and feedback”, present examples from a case study undertaken at Statoil. We also describe proof-of-concept prototypes that have been implemented to evaluate our approach.

In the final chapter, we summarize our work and contribution and consider directions for further work and research. We focus on three areas:

- 
1. Model-based design patterns: as a means of supporting movement in the design representation space described in Chapter 3, “Design and representation”.
  2. UML integration: how the RML, TaskMODL and DiaMODL languages may be integrated with UML
  3. Development tools: how the DiaMODL language should be supported by development tools, to combine the advantages of abstract user interface modelling and concrete design in traditional GUI-builders.





# Chapter 2

## State of the Art

In this chapter we give an overview of the trends in user interface modelling and model-based user interface design methods. Then we discuss two possible directions for making user interface modelling and model-based design more relevant and widespread for industrial use.

### 2.1 User interface modelling

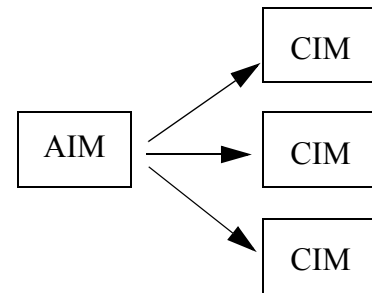
A possible definition of “user interface” is “all the *directly experienced* aspects of a thing or device”. With such a definition it is difficult to exclude aspects, highlighting the fact that almost everything contributes to the user’s experience of a product. For instance, the user interface of a screwdriver includes its shape, weight, grip and colour. The user interface of a computer in general comprises its *input* and *output devices*, like bitmapped screen, keyboard and mouse. Since a computer that is running an operating system and a set of applications is a *dynamic system*, its user interface includes the way it outputs information and interprets the user’s *actions* over time. From the point of view of the *end-user*, the dynamic state of the computer will limit the actions that may be performed, and this affects the *tasks* the user needs to perform in order to reach his *goals*. The extent to which the user is able to *understand* an interface will affect his or her *performance* when using it. From the point of view of the user interface *designer*, the user’s *world-view*, goals, and *abilities* determine which *functions* the interface should provide, which *dialogue* it should support and how this is achieved in terms of *concrete dialogue elements* for the chosen *platform*.

The highlighted words can all be considered to represent more or less directly experienced aspects of a device or considerations of the designer. In principle, all these aspect must be considered when performing user interface design. Various approaches will in practice put more focus on some aspects than on others. For instance, user-centred design (UCD) puts more emphasis on understanding the user, his environment and goals/tasks, than on guiding the design of presentation and behaviour for specific interaction techniques. In this work we focus on the use of *models* for capturing knowledge, ideas and suggested solutions for the interface design.

Historically, it seems that models in HCI belong to two different traditions, which we will call the *engineering* and *cognitive* traditions. The engineering tradition has its roots in formal methods within software engineering and implementation technology. In its infancy, the

focus of tools for user interface development was on making it practical to code the interface. Hence, toolkits with programming interfaces in various languages were developed, with more emphasis on flexibility and control than on ease of use. Higher-level frameworks were then built to make the developer more productive, with the most advanced systems hosted by powerful Smalltalk and Lisp environments. Then programming became more declarative and grew into powerful User Interface Management Systems (UIMS), with support for constructing and running complex interfaces. These tended to be self-contained and closed systems, without the control and integration provided by lower-level toolkits.

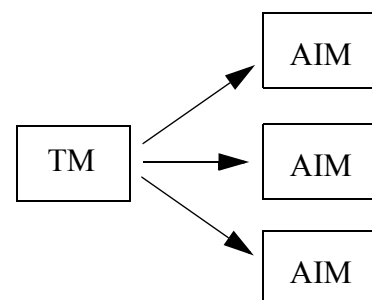
The declarative and abstract nature of UIMS languages can be seen as the forerunner of today's model-based tools and systems, e.g. Humanoid ([Szekely, 1990] and [Szekely, 1992]) and UIDE ([Foley, 1988] and [Sukaviriyaya, 1993]). The differences among them have to do with their foci: while the UIMS architectures and languages were designed to create executable specifications, modelling languages focus on providing support for analysis and reasoning. Later work on dialogue modelling and user interface architectures attempted to formalise previous work on *interaction objects*, and improve reasoning support, e.g. for mapping from abstract interaction models (AIM) to concrete interaction models (CIM) as illustrated in Figure 3.



**Figure 3.** Alternative concrete interface models (CIM), based on an abstract interface model (AIM)

The second tradition, labelled *cognitive*, is based on cognitive psychology and the work on formalising human cognition and behaviour. Its roots [Olson, 1990] are the Model Human Processor (MHP) [Card, 1983] and the GOMS [Card, 1980] formalisms, which provide a frameworks for evaluating how humans behave and perform when using a particular interface design. In a GOMS model, task performance is analysed in terms of *goals*, *operators*, *methods* and *selection rules*, structured in a hierarchy. GOMS models include mental operations and can be interpreted as programs executed by our cognitive machinery, as modelled in MHP or other cognitive architectures (see [John, 1996] for a comparison of various GOMS methods).

The two traditions meet in their interest in task analysis and modelling, and many of the GOMS ideas live on in contemporary task modelling approaches. TKS [Johnson, 1991] is an early attempt at bridging the gap between cognitive science and engineering. [Markopoulos, 1994] introduced the LOTOS process algebra notation as a formalisation, upon which the later ConcurTaskTrees (CTT) [Paterno, 1997] is also based. The interest in task modelling in the model-based approach seems partly to be due to methodological shortcomings of standard engineering approaches to design: Although functionally complete user interfaces may be built and even generated from e.g. object models [Balzert, 1995], they will not be very usable unless the users' tasks are taken into account. Hence, it is important to introduce task knowledge into the development process. ADEPT [Johnson, 1993] was an early demonstration of a visual tool for task-based interface design in the model-based tradition. ADEPT supported both task and abstract



**Figure 4.** Alternative abstract interface models, based on a task model.

interface models and mapping from the former to the latter (see Figure 4). Although similar to Figure 3, going from task models to abstract interaction models is far more difficult than going from abstract to concrete interface models. Abstract and concrete interaction objects (AIOs and CIOs) are essentially the same things at different abstraction levels, both representing a component of the artefact domain, while tasks are something actors perform, not part of the artefact itself, and may be both observed or designed.

Kind	Content	Representation
Task	what the user does or wants to do and why	There are two main kinds of task models: hierarchical: order/sequence-oriented models and process/data flow models. The former stresses the constraints on the sequence of subtasks in a hierarchy, the latter how data is used in tasks for control and processing. Task models are used both for describing <i>current practice</i> , to judge potential for improvements, and for <i>envisioning</i> or <i>prescribing future</i> tasks for which a new user interface is to be designed.
Domain/data:	concepts, object and operations etc. describing the domain	Tasks use and operate within and on the domain. Main formalisms are entities with attributes and relations, and objects with attributes, relations and operations and aggregation hierarchies.
Dialogue/conversation	the structure of dialogue between human and computer.	Behaviour-oriented abstractions like interactors, Petri Nets, flow charts, UML Activity and Sequence Diagrams, and state transition diagrams are used.
Concrete interaction	details of interaction, usually split into presentation and behaviour as follows:	
- Presentation	the structure of output and mapping to data and operations	Usually represented by generic but concrete dialogue elements and graphics
- Behaviour	means of inputting data and invoking operations	Describes how the dialogue is driven by the user's (physical) interaction and includes the link between presentation and dialogue, such as clicking a button to pop up a viewer.
Control	the application's services that task performance rely on	Usually a list of functions or object operations that may be invoked, and their pre- and post-conditions. Can be used as functional requirements for the application.
Platform	input and output capabilities of the device	Attributes and qualities may be defined, which can be referenced by interaction object mapping rules.
Environment	the physical and cultural context of interaction	Informal descriptions
User	characteristics and abilities of end-user not captured by other models	Classification hierarchy of user stereotypes, containing attributes and qualities that can be reference by e.g. mapping rules.

**Table 1.** Kinds of user interface models

The two mappings, from tasks to abstract dialogue to concrete dialogue, as well as *evaluation*, are the main steps in a model-based and iterative design process ([van der Veer, 2000] and [Paterno, 2000b]). In addition to task and dialogue models, these three steps rely on additional supporting models that capture knowledge of both the problem and solution

domain. [Vanderdonckt, 1999a] lists nine different (kinds of) models that may be used in interface design, as shown in Table 1.

As indicated in the table, the main steps from tasks and domain models, via abstract interaction or dialogue, to concrete interaction, are supported by models about the application core, target platform, working environment and user characteristics. The actual content and requirements of all these models depend on how they are used within the *design process*, and hence are somewhat ambiguous. For instance, a task model targeted at automatic generation of interface architecture will need to be more formal and complete than one focusing on mutual understanding among human participants in a design group. The focus of a method will also implicitly depend on the tradition, engineering or cognitive, within which it was conceived. Although there may be consensus concerning the relevant models, the details of a method may vary considerably.

Early tools for model-based design focused on *generating* all or parts of the user interface, either based on hard-coded algorithms, as in JANUS [Balzert, 1996], or knowledge-bases containing mapping rules, as in TRIDENT [Vanderdonckt, 1993] or design templates, as in Humanoid [Szekely, 1990]. A large body of rules for selecting concrete dialogue elements based on domain data was developed in the TRIDENT project, but a problem of this approach is managing the knowledge and handling specific domains [Vanderdonckt, 1999b], as well as scaling to larger dialogue structures. The work of Puerta has gone through an interesting transition from a mainly *generation* approach in MECANO [Puerta, 1996], via less emphasis on automation and more on *design support* through management of design knowledge with MOBI-D [Puerta, 1997], to integration of the model-based approach in an interface builder in MOBILE [Puerta, 1999], with more control given to the designer. It seems that two motivations are working together:

- the limitations of automating design by using model-based reasoning are handled by giving the designer more responsibility and work
- the constraints of a top-down model-based approach are relaxed to liberate the designer

Implicit in both is the fact that user interface modelling and model-based user interface design have not reached the mainstream software developer. One possible answer, the *engineer's*, is that we need to make our languages and tools more formal and with more reasoning power, so *more* of the design task can be given *stronger* support. After all, design is reasoning about the problem and solutions domain, tasks and information, dialogue and interaction, respectively. A second answer, the *designer's*, is that here are human and organisational obstacles to the use of formal methods in user interface design, so we need different tools, tools that provide better support for the creative aspects of design. Since humans do not actually think in terms of abstractions and formalisms, we should build more flexible and concrete languages, as well as tools with better usability. The *practitioner* would perhaps give a third answer, that we need languages and tools that are better integrated with industrial methods for systems engineering and development, e.g. user-centred design and object-oriented analysis and design. The last two answers will be treated in the following sections.

## 2.2 Obstacles to formal interface design and development

The limited acceptance of the model-based approach to interface development may be a sign of fundamental problems of using formal approaches in interface design in particular, and systems development in general. This section will review and discuss the critique and suggest alternative approaches, with a focus on the role of design representations as a medium for communication and capturing of design ideas.

[Hirschheim, 1989] presents two dimensions along which development approaches are classified, giving four different categories. The epistemological dimension concerns whether we view the domain (including human affairs) as something we can get *objective* knowledge about, or view it as *subjectively* interpreted and socially constructed. The ontological dimension concerns whether the social world is characterized by *order* e.g. stability and consensus, or *conflict* e.g. change and disagreement. The four resulting categories are named *Functionalism* (objective, order), *Social Relativism* (subjective, order), *Radical Structuralism* (objective, conflict) and *Neohumanism* (subjective, conflict). With respect to the first dimension the role of a representation is to capture knowledge about the *problem* domain, that is, what is covered by the users, tasks and objects perspectives. Formal representations are targeted at consistent, complete and precise descriptions, which are characteristic of the objective view. The ambiguity, incompleteness and inconsistency of the subjective view are poorly supported, and the analytical power is correspondingly reduced. For instance, different views, whether ambiguous or inconsistent, has to be formulated in *separate* models and *incomplete* models are more difficult to analyse. With respect to the order/conflict dimension the design representation's support of the development activity is important, as this dimension concerns dialogue, power relations and conflict resolution. Formal representations are considered hard to read and write, and support only those educated and trained in them. Thus, they can be viewed as conserving power and considered less democratic. It could be argued that formalisms are useful for discovering conflicts, and hence helpful in an environment where consensus is valued, although the process of conflict resolution is not directly supported. In summary, formal representations used as a medium for human-human communication, seem most appropriate for the Functionalism approach, while in Social Relativism it may be helpful as long as the dialogue is not hindered.

[Bansler, 1989] divides development approaches into three *traditions*: System theoretical, Socio-technical and Critical. The System theoretical tradition views the domain including organisation as a cybernetic system and attempts to use engineering methods "...to rationalise work processes by introducing computer based information systems...". As mentioned above, many of the existing formalisms for user interface modelling are descendents from this tradition, although the scope and details have changed. The Socio-technical tradition views organisations as composed of interrelated social and technical systems, and has greater focus on human behaviour and needs. The Critical tradition focuses on the inherent conflict that exist between different groups like workers and managers with opposing interests. These three traditions correspond in many respects with the Functionalism, Social Relativism and Radical Structuralism categories of Hirschheim's, respectively, and the reasoning about formal vs. informal representations is similar. However, one of the crucial points of the Socio-technical tradition is that the technical system should not be optimized at the expense of the social, but rather balance the focus. This suggests that formal represen-

tations are problematic only to the extent that they cause a shift of focus from the social system, i.e. a bias towards the technical system. Many of the current formalisms are targeted at analysing human activities and behavioural patterns, with the aim of supporting these in the user interface design, so this may not necessarily be a problem. However, the point that the formal representations are difficult to validate for many stakeholders, still applies.

[Floyd, 1987] presents two paradigms of systems development, what she calls the *product-* and *process-*oriented views. The product-oriented view is characterized by its focus on representations of the product, and the development of a program is a "...controlled transition between increasingly formalized defining [representations] leading eventually to computer-executable code..." The formal representations are used to ensure that the transitions are correct, so this view not only allows the use of formal methods but heavily relies on them. Although this was a natural reaction to the software crisis, the problem with this approach, according to Floyd, is that the product development is de-coupled from the users' work environment. Part of the problem is that the formal representations make it difficult to support learning and communication between developers and users, and validation by the users. As a reaction, the process-oriented view is designed to complement the product-oriented one by relating program development to the work environment. Programs should be designed within and through its usage, to ensure relevant functionality and usable design. While Floyd's distinction is similar to the objective/subjective dimension of Hirschheim's, she seems to be more focused on end-users rather than end-user organisations. This suggests that using formal representations of user interface design is even more problematic than for systems in general, because its need for validation by end-users is greater. I.e. if system functionality is validated through the user interface representations, these must be accessible to the end-user. Hence, concrete and informal representations like sketches and prototypes should be used and not abstract and formal diagrams or models.

The use of end-user oriented prototypes for validation is typical of Participatory Design and User Centred Design, which can be considered process-oriented approaches. By making use of concrete design representations, end-users are enabled to validate both the abstract system functionality and concrete design, either by direct participation in the design process or through user testing. In addition to resulting in more effective and usable systems, these approaches reduce end-user resistance to a new system, and possibly increase workplace democracy, since the affected stakeholders may have a greater feeling of control of the process and ownership of the product [Bjerknes, 1995]. Efficiency, usability, and acceptance are practical and pragmatic arguments that are compatible with the Social Relativism and Socio-technical paradigms. From such a point of view, the use of formal representations to increase effectiveness and usability should be more acceptable, as long as it does not conflict with the use of informal ones. In a complementary approach, informal representations may be used for dialogue and validation and formal methods used for analysis and verification. On the other hand, the democracy argument which is characteristic for the Radical Structuralism and Critical approaches, can be viewed as incompatible with formal representations of functionality and design, as these are unlikely to be understood and appreciated by end-users.

## 2.3 Informal vs. formal representations

As the discussion above indicates, the conditions for using formal representations vary considerably with the chosen development approach. Structured *engineering* approaches are focused on systems building favour use of formal representations, since they support the analysis and transition to implementation that is needed for making effective, efficient and reliable software. Approaches that rely on participation of people unskilled in formal methods, naturally favour informal representations, as they enhance the dialogue with end-user and support validation. Unless political arguments are used, it should be possible to combine formal and informal representation, to get the benefits of both. In the end, the system description will have to be executable, so a formal representation must eventually be derived from the informal ones. Another important argument for formal representations is that of scale, to bridge the gap between stakeholders at higher organisational levels, some formalisation is desirable to ensure proper understanding and agreement.

[Ehn, 1993] discusses participatory design and suggests an approach of *design-by-doing*, while acknowledging the power of systems thinking and rational methods. He stresses the importance of involvement of end-users and developers in each others work, i.e. developers must enter the end-users work environment to appreciate and learn their skills, while end-users must participate in the design activity. In the *language-game* of design which he proposes, design representations are not used as *models* of proposed designs, they are rather viewed as *tools* for discussing and *envisioning* possible designs. If design representations are to be used as “typical examples” and “paradigm cases” they must be effortless to relate to the current work and design context, and hence should be concrete and informal and not abstract and formal models. [Arias, 1997] discusses, in the context of urban design, the use of both *physical* and *computational* media in design, and the strengths and weaknesses of both. Physical representations have the advantage of being natural and intuitive to manipulate, and may easily become an implicit part of communication. Computational representations are more dynamic and visualise behaviour better, while being less intuitive and invisible in use. *Concrete* and *computerised* representations should be able to inherit both these sets of advantages, although the usability of tools still represents a problem. The conclusion of [Arias, 1997] is that they should be synergistically combined. It seems that we have two different and complementary uses of design representations: that of stimulating the creative design process or language game, with a focus on aiding communication and that of supporting software engineering, with a focus on reflection, analysis and transition towards an executable system. The natural conclusion is to use informal representations for the former and formal ones for the latter.

The advantages of informal representations include their visual and concrete nature. Although most suggested user interface modelling languages are graphical, they tend to be quite abstract. The *roughness* of many informal representations seems to be considered an advantage. [Landay, 1995] stresses that the roughness of a pencil sketch is important and that the lack of detail is a key feature of such sketches: “What designers need are computerized tools that allow them to sketch rough design ideas quickly”. [Gross, 1996] similarly identifies ambiguity/vagueness, abstraction, precision and uncertainty/commitment as important characteristics of design (representations): “[design tools] should capture users’ intended ambiguity, vagueness, and imprecision and convey these qualities”. Formal models, on the other hand, focus more on being “concise”, “complete” and “final” representations. Both Landay and Gross have made pen based systems, SILK [Landay, 1995] and The

Cocktail Napkin [Gross, 1996] respectively, that recognize pen strokes according to a pre-defined domain language. The key characteristics of sketches are handled in two ways: First, when trying to recognize the intention of the pen strokes, they keep track of alternative interpretations and are careful not to jump to any conclusions. Second, even after the interpretation is narrowed down to one possibility, the pen strokes may be kept in their fuzzy form. This ensures that the intended ambiguity, imprecision and uncertainty remains available for the reader to sense. There are few possibilities for expressing these characteristics in formal diagrams. The problem is that 1) diagram languages lack support for explicitly *omitting* syntactic elements and for encoding the *process state* of the design, and/or that 2) tools for drawing diagrams lack this possibility. There is an important difference between explicitly expressing few constraints, and not expressing any. Consider a data modelling tool where the default cardinality of a relation might be set to “zero-or-more”. Logically this may make sense, since this is the least constraining value. However, from the modeller’s point of view, there is a big difference between “undecided” and “unconstrained”, and this information is lost when the value is filled in by default. Note that this problem is not specific for diagrams, also GUI builders makes it difficult to be vague. For instance, when creating or resizing a dialogue box its dimensions cannot be set to “big”, it will be fixed to the one drawn. It may seem that the focus on helping to make syntactically complete and correct representations, actually works against the designer.

[Landay, 1995] cites an observation that “rough electronic sketches kept [the designers] from talking about unimportant low-level details” and that “[the] finished-looking interfaces caused them to talk more about the ‘look’ rather than interaction issues.” The intention put into the rough sketch or prototype and the polished one may be the same, but for the dialogue among developers and with the end-user, the former is more stimulating and less constraining. [Jones, 1998] notes that informal representations “has the advantage that it does not impose any conceptual constraints on those drawing”. However, to ensure that agreement is really based on a proper (and common) understanding, it is suggested that more formal representations are used, especially in the later stages of design, when it is important to be able to share the design with people outside the context within which the design was conceived. Such communication is necessary in the larger context of system development and engineering.

## 2.4 User interface design within a systems development context

The user interface design activity will usually be part of a larger systems development process, the goal of which is to produce both the *back-end system* and the *user-visible tools*. As argued in Chapter 1, “Introduction”, this distinction is usually irrelevant for the end-user, but important from a development and engineering perspective. There are several reasons for integrating system and interface design and modelling, e.g:

- the end-user organisation may require that certain features of an existing “legacy” system be supported and remain unchanged in certain ways,



- system specification may run in parallel to the user interface design activities and hence must be coordinated with it,
- the user interface design must eventually be integrated with the rest of the system specification, and
- at the least, the user interface must be implemented by engineering tools

The difference between these is the direction of requirements between system and user interface. In the first case, the system forces the user interface design to satisfy certain constraints, the second case is more symmetric and allows more negotiation, while the user interface design in the third case may dominate. In all cases, it is vital that the design representation can be compared with the system specification, and inconsistencies be identified. In addition, a more constructive use of representations is possible, where one is directly used as a starting point for the other. All in all, it would be an advantage if design representations in some way were aligned with system representations, e.g.:

1. the same modelling languages are used, e.g. the various sub-languages of UML,
2. an integrated family of languages is used, divided into perspectives in the same way as UML, i.e. languages for concepts, functions, behaviour and architecture
3. it must at least be known how to find inconsistencies and incompletenesses between design and system representations

The interest in the first and second option is growing, e.g. the TUPIS workshop [TUPIS'2000] discussed integration of user interface modelling with UML, and the CADUI'2002 conference has a *uniform user interface modelling language* as a special theme. The second option implies an integrated set of concepts for the nine models listed in Section 2.1, while the first option means interpreting or using the UML languages within the domain of user interface design. Although UML is not the only family of modelling languages used for systems development, it serves to illustrate the possibilities and problems of integrating user interface design and systems engineering from a language point of view.

### 2.4.1 UML and interaction design

UML [UML, 1998] is a family of languages based on a unified set of concepts described in a meta-model. Several diagram types are defined with a concrete syntax or notation for the abstract (meta) concepts. UML includes a light-weight *extension* mechanism, as an alternative to augmenting the meta-model. Sets of related extensions can be used for defining *profiles* for specific domains, such as interactive systems. UML provides diagrams for static domain modelling (class and object diagrams), functional requirements (use case, sequence and activity diagrams), behaviour (sequence, collaboration, state and activity diagrams) and deployment (component diagram). These could in principle be made to cover the nine sub-domains from [Vanderdonckt, 1999a] in the list above, but it is an open question how easily and well UML cover them.

[Markopoulos, 2000a] discusses how modelling of domain, tasks, high-level dialogue and low-level dialogue is supported by UML. Domain modelling is directly handled by class and object diagrams. Task modelling is a natural target for the functionally oriented dia-

grams, which include use case, sequence and activity diagrams. A use case describes a *class* of sequences of actions that a *system* may perform when interacting with an *actor*. An accompanying textual scenario often provides a more concrete description of a use case *instance*, while a sequence diagram may detail the sequences. Both [Markopoulos, 2000a] and [Constantine, 2001a] note that the traditional use of scenarios within interface design, that of capturing the context, goals, intentions, values of a user and the use of artifacts, is quite different from how use case diagrams and scenarios are used. The focus of the former is individual users, while the latter focuses on the system and multiple actors. Hence, instead of capturing the user's goals and how the user intends to reach them by means of the user interface, it models the system interface and the required system functions. Hence, use cases are not designed nor useful for understanding and identifying with the user.<sup>1</sup> This is consistent with a general problem of UML: the focus is on technically-oriented design, including architectural and implementation issues [Nunes, 2000a], not human-centred specification of functionality and behaviour. By including design-oriented concepts, like operations and methods and mixing attributes and relations, it becomes less suitable for modelling of real-world domains, whether object-oriented or not [Embley, 1995].

Both sequence and activity diagrams are behaviour-oriented languages and hence, are candidates for modelling the sequence of tasks. Sequence diagrams show how a group of actors or objects operate and communicate over time. When used for detailing use cases, the actors are either users or system functions, and communication is interpreted as interaction, information flow and/or message passing. Later, in the design phase, the sequence diagram is given a more concrete interpretation, that of describing the life-cycle of objects, and how they invoke each other's services or methods. [van der Veer, 2000] suggests that we reuse the visual notation of sequence diagrams in the context of task modelling. The actors in such a "task sequence" diagram would be users or roles and the system as a whole, while operations and message passing would be interpreted as task activation and information flow. Although the sequencing will be visible, the focus will be moved from task structure to actor interaction, which is only part of what task models should cover. A more fundamental problem is the way the visual notation is borrowed, without integrating the underlying meta-concepts of the sequence diagrams, i.e. actors/objects and messages/operations, with the task concepts. The other candidate for task modelling, the activity diagram, is also used for both analysis and design, e.g. for modelling business processes or control flow, respectively. [Markopoulos, 2000a] notes that activity diagrams "tend to bring in excessive detail", perhaps because activity diagrams are more focused on *executable* behaviour than on *observed* human activity. However, the biggest problem of activity diagrams, which it shares with sequence diagrams, is its lack of support for hierarchies, and hence, hierarchical task structures.<sup>2</sup> Instead, the model must either be flattened or split into separate diagrams to explicitly indicate nesting. A possible way of realising the latter option, is to model tasks by using objects that are nested by means of UML's aggregation relation. Each activity diagram will be part of the class definition of an aggregated class. Unfortunately, inheritance of activity diagrams of classes in hierarchy is not well-defined in UML [Bergner, 1998]. In summary, task modelling is not particularly well supported by UML.

---

1. Note that this is a problem shared with most process-oriented modelling languages like Dataflow Diagrams (DFD): the user is just another external entity at the boundary of the system.

2. This may perhaps come as a surprise, since activity diagrams are defined in terms of Statecharts, which are hierarchical.

Given UML's focus on the specification of software artefacts, it seems better fit for modelling interaction objects, both abstract and concrete. The Statecharts language [Harel, 1987] is a formalism supporting hierarchical state machines, and UML includes a variant of Statecharts with a somewhat more practical notation than the original.<sup>1</sup> Statecharts is designed for describing the behaviour of *reactive* systems, of which modern event-based user interfaces is a kind, and has been used for implementing a UIMS [Wellner, 1989] and for user interface design [Horrocks, 1999]. The strength of Statecharts is the support for hierarchical states, but this hierarchy may unfortunately not cross object boundaries. A hierarchical object structure where each object's life-cycle is described by a state machine, can be considered a large composite state machine. But each object's state machine is isolated from the others', and must communicate through the standard object-oriented mechanisms of relation traversal and method invocation. [Markopoulos, 2000a] suggests using Statecharts for high-level navigation and activation of dialogues, but it is unclear if Statecharts could also be used for the internals of each dialogue. This seems desirable and feasible, since it will make the interaction model more uniform. This will however, require a suitable representation in UML of the static structure of interaction objects and their relations, like hierarchical containment, data values, and value constraints, for which the class diagram is a natural candidate.

If UML does not provide direct support for a user interface modelling perspective or language, it may be extended by means of its stereotype mechanism. This mechanism provides a way of deriving new model element classes or meta-concepts from the existing meta-classes in UML, such as Classifier and Association. Model elements may be marked as being an instance of this stereotype, to highlight that the standard interpretation of the element must be augmented by the stereotype's. The WISDOM approach [Nunes, 2000a] (see [Nunes, 2000b] for details) uses this mechanism to extend UML with a family of user interface modelling concepts. With the <<Entity>>, <<Task>> and <<Interaction space>> (view) *class* stereotypes, the UML modeller can build user interface models including domain objects, tasks and structures of presentation and interaction objects. *Association* stereotypes are used to relate elements in ways specific for the participating stereotypes, and rules may constrain the allowed associations. For instance, task and presentation hierarchies are modelled using the <<Refine task>> and <<Contains>> (aggregation) association stereotypes, respectively. <<Navigate>> association stereotypes are used between <<Interaction space>> model elements, to express how the user can move among them. WISDOM includes ConcurTaskTrees in its extension of UML by providing stereotypes for all its constructs, including sequence constraints. The main problem is that although the abstract syntax of ConcurTaskTrees is covered, the semantics of the underlying LOTOS process concepts cannot be expressed. Instead, the new constructs may inherit features from their base meta-class, that may not fit or have a natural interpretation. For instance, the <<task>> stereotype must allow methods to be added and sub-classes to be defined, since it is a *class* stereotype. Finally, although the main reason for extending UML is to ease the integration of new sub-languages, the concrete syntax or notation cannot easily be extended. Hence, the usability of the integrated language suffers.

In [da Silva, 2000] a similar approach is presented. Existing user interface modelling languages are integrated with UML's standard diagrams, by means of stereotypes. However, in a parallel effort, UML's own semantics is defined in terms of LOTOS [da Silva, 2001]. This

---

1. However, no tool we know includes full support for UML's Statecharts dialect.

provides UML with a more complete and formal meaning, supports retaining the original semantics of LOTOS-based modelling languages and enables formal analysis of the integrated languages. This approach has its own problems, e.g. gaining acceptance for a competing semantics, but it is an interesting way of handling the semantic mismatch between UML and user interface modelling languages.

## 2.5 Summary

In this review we have identified several problems of current approaches, some of which we hope to target in our own work.

- The modelling languages should be based on few and simple basic concepts and constructs, to make them easier to read and write. The languages should not force the modeller to be more specific and formal than the development process requires. (Section 2.2).
- The languages should not force the modeller to include details before they are actually needed. They should support a gradual transition from general to more specific and formal statements. The visual notation should be flexible (Section 2.3).
- There is a need for combining task models with models of the user interface's structure and behaviour, while still keeping them separate conceptually (Section 2.1).
- Tools for modelling the user interface's structure should build on the functionality of existing tools for "drawing" user interfaces, and hence the models of abstract interaction objects should provide a natural transition to and from concrete interaction objects (Section 2.1).
- The formalisation of the modelling languages should allow for integration with existing languages used in software engineering (Section 2.4).

All these problems can all be considered integration issues. The first two list items concern integrating along the design process activities, as both of them target the level of formality and detail of models (or design representations in general). The third and fourth list items concern integration across sub-domains, -languages and -models; the third the relation between and integration of task and (abstract) dialogue and the fourth abstract dialogue and concrete interaction. The last list item concerns integration across user interface design and systems development.

Tackling these issues requires a deeper understanding of how design representations in general and models in specific are used. In the following chapter, Design and representation, we will therefore introduce a framework for classifying design representations. These issues will be discussed and reinterpreted in this context. Chapter 4, "Task modelling"; Chapter 5, "Dialogue modelling" and Chapter 6, "Concrete interaction" will all return to these issues in the context of the classification framework.

## Chapter 3

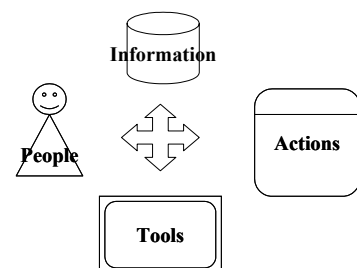
# Design and representation

In this chapter we present a high-level view of information systems, with the main elements that engineers and designers must relate to and understand when building systems. We place ourselves within the engineering tradition, and note that the systems engineering and user interface designer traditions represent different approaches to information system development. This tension must be taken into account when designing methods and tools for developing information systems, so that these are usable also for non-engineers. Although the use of models distinguishes these traditions, we suggest that diagram based models may nevertheless be used to integrate them. We then take a closer look at representations used when designing user interfaces. Representations with different characteristics have different roles and can be used for different purposes. We introduce three dimensions for classifying these representations and use this classification in a discussion of the role of representations in the design process. Formal representations can play only some of these roles, and hence must be integrated with other representations.

### 3.1 High-level view of People, Actions, Information and Tools.

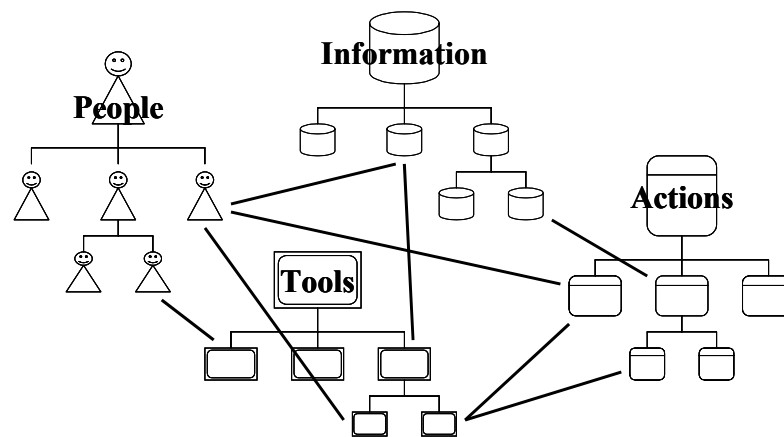
Human beings are distinguished by intellectual abilities for information processing and knowledge creation and management. We not only use advanced tools, but build advanced tools and build tool-developing tools. Both the information and the tools are used for performing actions, both individually and collectively, as illustrated in Figure 5. When designing tools it is crucial to understand the relations among this quadruple, whether the tool to be designed is a Personal Information Manager (PIM) hosted by a Personal Digital Assistant (PDA) or an Enterprise Resource Planning (ERP) system accessed through specialised clients or web browsers.

For instance, it is important to know why people perform certain actions, how people use information and tools to perform these actions, how tools can represent and process information, and last but not least, what additional support people can be given for performing actions by new tools.



**Figure 5.** *People use information and tools to perform actions*

The figure illustrates that these entities, i.e. people, information, tools and actions, are all related. For instance, people perform actions in the pursuit of goals, they seek information that these actions require, tools present and process information and achieve goals. These entities can all be considered complex entities, with structure and internal relations, as well as relations among each other. People combine into organisations, information is collected in databases, tools are constructed out of components and actions are part of activities and processes. The simplest model of these structures is the hierarchy, as illustrated in Figure 6, but a hierarchy reveals only part of the picture. Although an organisation may have a formal hierarchical structure, important relations like communication, power and authority may cross the hierarchical boundaries. Similarly, information can be extensively cross-linked like in the World Wide Web or related through arbitrary relations as in relational databases. Triggering or enabling of actions in a process may follow complex patterns, and the supporting tools may be even more complicated. Part of the problem of designing useful tools stems from this complex relationship. In addition, a single tool must often support a heterogeneous user population in the performance of many different actions, as part of processes at many levels.



**Figure 6.** *People, information, tools and actions are complex entities*

Many different disciplines have approached the problem of understanding this complex picture. Different disciplines and fields study different parts of the picture, different relations among the entities and at different levels in the hierarchies, as suggested in Figure 7. Although it is impossible to be fair to any discipline within such a simple framework, it may be enlightening to place them in this concept space. In the figure we have placed sociology and psychology by the people structure, since that is their primary focus. This does not mean that they are not concerned with action, information and tools, but that these are viewed from people's point of view. Similarly, semantic data modelling primarily study the structure and nature of information, but is also concerned with actions upon data. Other disciplines study several entities and their relations. For instance, functional analysis is concerned with both information, action and how these are related. Similarly, the figure suggests that the object of study in Business Process Reengineering (BPR) is the relation between organisation and their use of information for reaching their business goals and driving high-level processes. Some disciplines differ mainly on the level of focus. For instance, task-based user interface design may simplistically be viewed as the study of how people use computer-based tools for reaching goals and performing actions, with a primary focus on small groups and individuals and the applications they use. This is analogous to

what a workflow study is about, although the goal of applying a workflow view is more limited and the focus is on groups, application suites and processes.

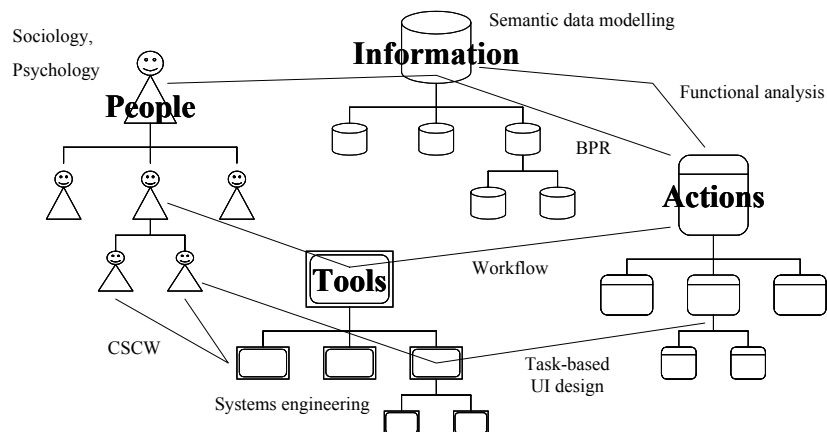


Figure 7. Different disciplines study different parts of the picture

This high-level view reveals boundaries that exist among entities and levels. This is important, since once the boundaries are identified and conceptualised, they become better targets of investigation. For instance, usability is related to the separation of people and their tools. The user’s abilities and expectations may differ from the tool’s requirements and actual behaviour, and this may lead to usability problems. The classical difference between system analysis and design is concerned with the difference between what we want to do with a tool, i.e. functional requirements, and the construction of the tool itself, i.e. its design.<sup>1</sup> Boundaries may also exist between entity individuals and the levels within each entity hierarchy. People may have different world-views, goals and opinions, and tasks performed by them may not always, when composed into processes, contribute to the collectively stated goal. Information sources may use different names for the same concepts (synonym) and the same for different ones (homonym), and user interface components may be incompatible and impossible to integrate. These are all important problems to be aware of and to study.

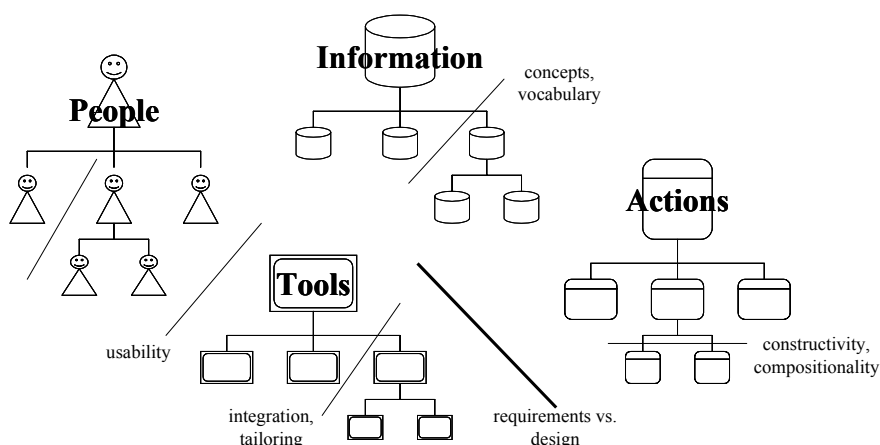


Figure 8. Boundaries are interesting and difficult to handle

1. The word “design” is used in two different ways, either meaning technically oriented construction as here, or as the specification of the behavior and form of a user interface.

Some boundaries are introduced by limiting a discipline to the study of only part of this high-level view, and can be a source of critique and conflict. Systems engineering may be criticised for ignoring the organisational and human aspect of information systems, while sociological approaches on the other hand give little guidance for how to build flexible, scalable and robust systems. The cognitive and designer traditions discussed in Chapter 2, with their focus on humans, have traditionally been in opposition to the engineering tradition, with its focus on implementation technology, rather than on the people and the activities that the technology is supposed to support. Hopefully, by being aware of these boundaries, we can both prepare ourselves to handle the problems caused by them, and utilize research results from different disciplines. The mentioned difference between workflow and task-based design may for instance be used constructively, by basing a task analysis on the workflows that the relevant user groups participate in, as suggested in [Trættestad, 1999].

## 3.2 Engineering and designer traditions

Today's structured methods for developing information systems, can be considered an answer to the "software crisis", which was the name given in the late 70's, to the problems experienced with developing complex software. The remedy was to turn software development into a structured engineering discipline, where the overall goal was to develop high-quality software for supporting information systems, at a lower cost and resource consumption and with predictable results. Although the engineering approach has been successful for constructing the software, it is not known for delivering good user interfaces. It seems that designing *usable* software is different from designing *functionally* sound and complete software [Kim, 1995].

One of the main motivations for this work is to extend the engineering tradition into the domain of user interface design. Our approach is to do this through integration with existing non-engineering methods, instead of replacing these. Engineering methods should and will be used side-by-side with traditional designer methods. Since proper integration requires understanding both of the approaches and in particular appreciating the differences, we will try to characterise and contrast the approaches. We have deliberately exaggerated the differences, so the description may be seen as caricatures of how they are really practised. Note that we use the terms "design" and "designer" in a restricted and specific sense here, as the activity of conceiving the appropriate user experience and shaping the user interface accordingly, and as the developer who performs this activity. The second established usage of "design", as in "software design", is instead referred to as "construction". The construction material is objects and processes, rather than directly experienced elements of concrete interaction.

Fundamental to the engineering approach is its use of formal methods. To better control the quality of the final product and the development process, these were introduced for

1. describing the problem of the customer organisation
2. specifying the functional requirements for the computerised solution and



3. ensuring that the solution's construction fulfilled the functional requirements, in addition to non-functional and architectural requirements.

The use of the term *engineering* (systems or software) is no coincidence, as the success of structural engineering is seen as both an inspiration and goal. Crucial to this field is the decoupling of deriving the required characteristics, constructing a solution according to the requirements and actually building it. Using a bridge as an example, first the required strength and agility would be derived from the required span and the load and weather conditions it should sustain. Second, based on the characteristics of the basic constructional elements like beams and cables and the behaviour of their composition, a construction for the whole bridge could be derived. Characteristics of the parts and the whole of the construction could be verified against the constraints of the requirements, based on mathematical methods. Finally, the bridge could be built with the confidence that the construction was sound, both technically and economically. With a formal description of the construction, it would be considered safe to let someone else build it on contract.


The engineering approach to systems development has a strong focus on specifications and descriptions of the problem and solution, and on methods for moving from the former to the latter. The approach is inherently top-down from problem to solution, since the formal descriptions derived in each of the steps above can be seen as a specification or a set of requirements for the next step. In addition, the specification is supposed to drive the development of the solution, in addition to being verified against it. *Verification*, i.e. checking that various representations of the same domain are consistent, increases the confidence that a solution is sound. The stronger and more complete the theory of the domain is, the less need there will be for *validation*, since more can be formally checked. There are limits to what a theory can cover, and hence, there will still be a need for iterating through problem specification and solution construction. However, basic assumption of the engineering approach is that with comprehensive modelling and careful analysis the number of iterations needed can be minimize, and hence the total development time.

The *designer* tradition constraints the engineering tradition in several ways. [Löwgren, 1995] suggests that there is a fundamental difference between the requirements of creative design vs. software engineering: "Engineering design work is amenable to structured descriptions and seen as a chain of transformations from the abstract (requirements) to the concrete (resulting artifact)." This is in accordance with the model-based design tradition, as presented in Chapter 2. In contrast: "Creative design work is seen as a tight interplay between problem setting and problem solving." Hence, the *medium* from which the designer builds the solution, becomes more important. The construction medium not only provides the potential for solving a stated problem, it also represents a way of interpreting and understanding the problem in the first place. Hence, understanding and experimenting with this medium during the development becomes very important, and goes hand-in-hand with understanding and specifying the original problem. The importance of experimenting with the medium affects how the designer tradition represents design suggestions. Although the representations can be seen as abstractions of the real design, they use the same visual language, perhaps with a different scale, fidelity and scope. Whereas engineers abstract away technical details to better concentrate on the essence of problems and solutions, designers prefer to deal with concrete representations, as representatives for more general solutions. In addition, designers prefer not to split the representation into parts covering different aspects like structure and graphics, since the design must be validated as a whole.

Another important difference is the role of *values*, which is used both when interpreting the real-world problem and for validating a solution. There is a strong solidarity with the end-user, which quite often is different from the customer paying for the development project. This *user focus* is different from the *system view* that is natural in the engineering tradition, and will often lead to designs that are more usable from the end-user's point of view, while not necessarily easier to manage or implement, from a development point of view. The User Centred Design (UCD)<sup>1</sup> and Participatory Design traditions require end-user participation in the development process that goes beyond that considered necessary in the engineering tradition, both for eliciting requirements and for validating the solution.

The major differences between the engineering and designer traditions give rise to a two-dimensional classification, based on formality of the approach and the focus, as shown in Table 2. We see that information systems engineering and the designer tradition occupy opposite corners of the matrix, system focus and formal approach and user focus and informal approach, respectively. The two remaining cells can be seen as two possible paths for bridging the gap between the engineering and designer traditions. From the information systems engineering cell, we can either 1) change the focus to users by using the approach of model-based user interface design and then decrease the formality of the approach, or 2) employ a less formal approach like extreme programming (XP), and then change the focus. XP is interesting, since this approach is driven by use case descriptions and is centred around on rapid delivery and validation of new iterations. This is similar to user-centred prototyping, although with a system focus. In this work, we take the first of these paths, i.e. we try to make the models and techniques of information systems engineering and model-based user interface design, easier to work with as a whole, and within the approach of the designer tradition.

		Focus	
		System	User
Approach	Formal	Systems engineering	Model-based UI design
	Informal	Extreme programming (XP)?	Designer tradition



**Table 2.** Classifying development approaches

### 3.3 Three dimensions of design representations

Design of user interfaces is a complex and difficult task. It is usually a creative and collaborative process, in which several participants must both generate design suggestions and evaluate the suggestions with respect to both functional and usability requirements. The goal of the user interface development activity is a user interface with a certain *usability*, while the primary product of the user interface design process is a specification of a user interface meeting this goal, from which an implementation will be engineered. It is natural

1. This acronym is sometimes used for the seemingly similar approach of *Usage-Centered Design*, which is better shortened to UsageCD.

to make and use many kinds of representations during this process, to make design knowledge accessible to the designers, to record the current state of development and to move towards the final specification.<sup>1</sup> The representation language can be seen as a tool for making this specification and must be able to capture the elements needed for designing a usable interface. [ISO 9241, 1997] defines usability as:

*“The effectiveness, efficiency, and satisfaction with which specified users achieve specified goals in particular environments.”*

It is clear that the developers need an understanding of who the user is (“specified users”), the environment she lives and acts within (“particular environment”), the goals (“specified goals”) she wants to reach and what task she therefore needs to perform. To be able to capture and preserve the understanding of all these aspects of a specific design problem, a designer will need to write it down in a suitable form, i.e. represent this knowledge. As mentioned above, various kinds of formal descriptions and specifications play an important role in the engineering approach. The same is true for the model-based approach to user interface design.

Table 1, in Chapter 2, shows nine kinds of models used in model-based design. The models represent different views or *perspectives* of the domain of user interface design, by covering different aspects of it. Some of these can be considered problem-oriented, since they describe requirements of the domain or of the goals we want to achieve with the design, while others are solution-oriented since they describe aspects of the artefact that we are designing and the environment within which the artefact will operate. We understand the differences between problem and solution to be:

- **problem/domain:** the characteristics of the different kinds of *users*, and the *structure* of the *work* or *tasks* the user wants to perform or will be performing, and the user’s view of the *domain*
- **solution/artefact:** the *structure* of the *dialogue* between user and system, and the *concrete interaction* which details the visualization and interaction

The distinction between problem- and solution-orientation is not always clear, as some models may be considered to be both a solution for a specified problem and a specification for a problem to be solved further on in the development process. For instance, an abstract dialogue model can be considered a solution to the problem of supporting a set of tasks, while at the same time representing requirements for the design of concrete interactions. Hence, we introduce a problem/solution dimension for classifying user interface design representations, as shown in Figure 9.



**Figure 9.** The problem/solution dimension for classifying design representations

1. In this context we are mainly interested in representations external to our mind. Although mental models are important to understand, it’s the external form we focus on here.

The three main user interface models, 1) tasks/domain, 2) abstract and 3) concrete interaction, can be placed along the problem-solution axis, as shown in Figure 11, as can the other models in Table 1, Chapter 2. For instance, the physical *environment* and design *platform* is part of concrete interaction, and may span several levels, from room layout, via desk design to input device ergonomics. One interpretation of this dimension is that it represents the extent to which the final user interface design is constrained, the number of constraints increasing when moving to the right of the axis. According to this interpretation, many different user interfaces can be designed to satisfy specific domain and task models, while fewer design proposals will satisfy the constraints represented in a solution-oriented dialogue model. Note that this interpretation does not suggest that the constraints are explicitly expressed, only that the different models provide different levels of freedom for subsequent design. Neither does it mean that the design process proceeds monotonously from left to right, although many methods seems to suggest or assume this.

The languages that have been suggested for modelling user interfaces, tend to focus on only one perspective each.<sup>1</sup> The task perspective focuses on work structure, the dialogue perspective on the life-cycle and activation of dialogue elements, and the interaction/presentation on the look & feel of the user interface. In the course of designing an interface, several languages have to be used to cover all the needed perspectives.

Within each perspective, a language will typically be used to model objects of various *granularities*<sup>2</sup>, e.g. as a hierarchy. A particular design representation may target only part of this hierarchy, and correspondingly a representation method/language may be designed and suited for only this part. For instance, a workflow modelling language is targeted at high-level, group-oriented tasks, while a task modelling language typically is targeted at individual lower-level tasks. The *granularity* of the objects described is our second way of classifying representations.

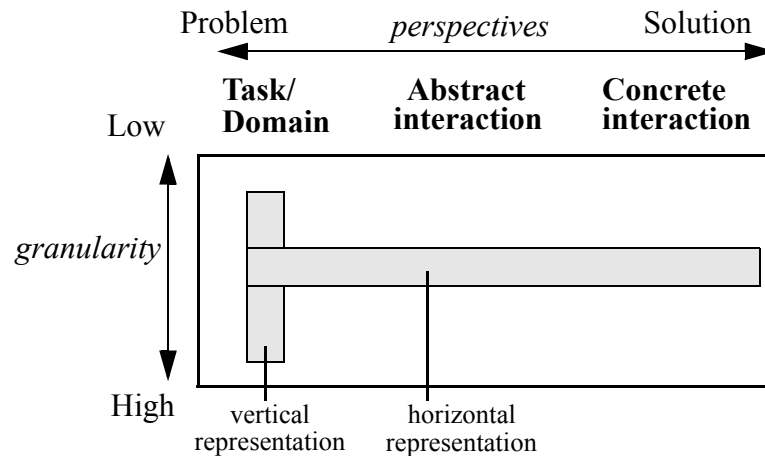


**Figure 10.** The level/granularity dimension for classifying design representations

Note that the level dimension is similar to the concept of abstraction, i.e. a high-level view is more abstract than a low level view. However, the concept of abstractness is also related to the perspectives dimension discussed above, e.g. a dialogue model is more abstract than a concrete interaction model, since it abstracts away lower level and often platform specific details. Therefore we will avoid using that term for classification.

The perspective and the granularity of objects gives a space of two dimensions, i.e. problem/solution and low/high granularity, as illustrated in Figure 11. Any particular design representation or modelling *method/language* will cover only part of this plane. As shown, vertical representations cover objects of one perspective at several granularities, which is typical for formal models. Informal representations like scenarios, sketches, snapshots and storyboards, as well as natural language, are usually not targeted at specific perspectives.

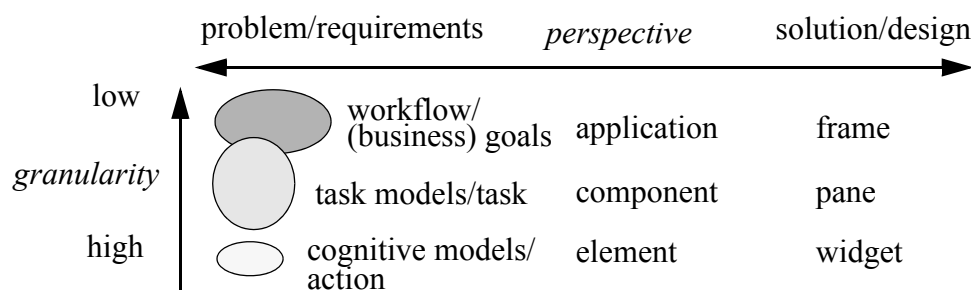
1. The advantage is that each formalisms can be tailored to the part of the domain it covers, the disadvantage is that the different sub-domains become more difficult to combine in one model.
2. We choose to use the word *granularity* instead of *level*, because the latter suggests that the points along the axis are well-defined and aligned across perspectives.



**Figure 11.** Design representation space: perspective and level

For instance, a storyboard can contain elements of users, tasks, objects and both abstract and concrete interaction. Since such informal representations may cover several perspectives at once, these are considered *horizontal*, as illustrated in the figure. The various levels may be treated differently by different languages, and may not be aligned across perspectives. Usually, a representation method/language family will use the same concepts across all levels of a perspective that it covers, and align levels across perspectives. For instance, a workflow language will typically use the same formalism for processes, activities and actions, the different terms corresponding to different levels. In addition, if the supporting tools are modelled by concepts like application suites, applications and components, these will likely be aligned with the work oriented ones. A task modelling language may be similarly structured, although its focus may only partially overlap in the granularity dimension.

Using the same concepts for describing a perspective across levels has many advantages, both practically and theoretically. For instance, the possibility of deriving aggregated characteristics for one level and comparing it with those specified by the level above, i.e. the compositionality/constructivity of the constructs, is easier/only possible when similar/equivalent concepts are used. Typically, the more solution-oriented a perspectives is, the more difficult will it be to treat the levels uniformly, since it will be more difficult to hide the perceived differences among design elements at each level. For instance, a toolbar button is sufficiently different from an application window to require different treatment, while both may be designed to support the same concept of task, albeit at different levels.

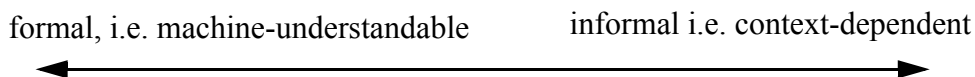


**Figure 12.** The level/granularity dimension interpreted across perspectives

Figure 12 suggests one interpretation of the granularity level across different perspectives. The main point is not the absolute placement, but rather the relative position within and

across perspectives. For instance, a task is performed to achieve a goal, and is often supported by a specific component<sup>1</sup> composed of dialogue elements, which is placed in a pane containing widgets. The figure also indicates how different task-oriented models cover the granularity dimension. As can be seen, workflow is mostly problem-oriented at a high level, with a touch of solution aspects. Task models are medium level, with a large span, and even more problem-oriented. Cognitive models fill the bottom of the granularity dimension, as it is mostly concerned with low-level perception and goal-setting.

The perspective and the granularity level dimensions, cover the whole of the domain, i.e. everything that should be represented about design can be placed within this space. The form of the representation is not considered, however, and we will capture this in a third classification dimension. The question remains what aspect of the form of the representation is crucial in our classification. Based on the discussion in the Section 3.2 we have chosen to focus on the level of formality of the representation. There are two aspects of formality we are interested in, related to machine and human understanding, respectively. First, there is the extent to which a development tool can understand the representation, analyse and manipulate it. This requires some formalised meaning that can be programmed into a tool and some structure of the representation that is accessible. The second aspect is targeted at human understanding, analysis and manipulation and is concerned with how the situation or context is used to give meaning to the representation. A completely formal representation does not depend on a specific context for correct interpretation, while an informal one can only be understood given the appropriate meaning-giving context[Nonaka, 1998]. Figure 13 show the *formality* dimension and the two characterising features at each end of the axis.



**Figure 13.** The formality dimension for classifying design representations

These two distinguishing features bring our attention to the difference between man and machine, the former having access to the real world and hence the context for “really” understanding, the latter having great power for syntactic processing but little more. When we contrast machine-understandable with context-dependent, we may seem to suggest that formal representations are not humanly accessible, since they are oriented towards syntactic manipulation<sup>2</sup> and that informal ones cannot be usefully manipulated by machines, since the meaning-giving context is inaccessible to them. However, one of the main points of this work is that, yes, humans can fruitfully utilise the power of formal representations, and others have argued for and built systems that are based on machine manipulation of informal representations. While full consistency and completeness analysis are beyond our abilities, we can still use formal representation for guiding our thoughts, and while machines cannot give meaning to informal representations they can still support structuring, filtering, viewing and navigating them. The key is understanding how the representation can be used by us and machines, and focus on the relevant reasoning and manipulation for both. This will be further discussed later when the development process is considered, in Section 3.6.

- 
1. The term “component” has been used for both atomic dialogue elements, task-oriented groups of elements or semi-autonomous mini-applications. Here we have chosen the second of these interpretations.
  2. Formal systems are by nature syntactic, semantic reasoning is done by means of symbol manipulation.

The desired level of formality for a particular type of representation is highly dependent on usage. It is possible to use a fully formal language informally. This happens when the formal meaning of a representation does not match the intention of its users, and the context of usage makes this apparent. For instance, a notation can be (re)used for a different purpose or domain than defined for, or a diagramming tool can be used solely for the graphical appearance, not for the underlying meaning (surface/concrete vs. deep/abstract meaning/syntax). The opposite situation is perhaps more relevant, that of using an informal representation formally. This typically occurs when the cultural context is enough to give meaning to a representation, i.e. social and cultural conventions that are taken for granted take the place of a formal definition of the representation language. For instance, a pencil sketch of a dialogue window may be perfectly understandable, although the exact notation differs from sketch to sketch. It might be argued that the notation really is formal, the definition has just not been written down. However, its users may not be able to describe the notation, even though they use it without ambiguity problems, and for any particular definition examples of perfectly understood but uncovered/illegal representations can be made. Hence, there really is no formal language, but rather a formal usage of an informal one.

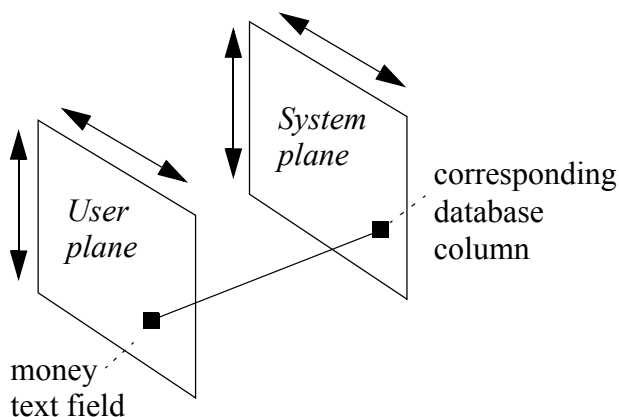
### 3.4 A fourth dimension: user - system?

The three dimensions introduced for classifying design representations can also be used for more system- and engineering-oriented representations. First, our distinction between problem- and solution-oriented perspectives was motivated by the discussion within engineering on analysis vs. design. Although it may be difficult to know the difference in practice, the distinction is conceptually important. Second, the languages and methods used within engineering are often hierarchical, or may at least be interpreted as containing levels. For instance, dataflow diagrams are nested, and although few databases support nested tables, they may be viewed as three-level entities consisting of logical databases, tables and fields. Third, methods and languages have varying degree of formality, and the level of formality may even vary within the same method. For instance, UML use case diagrams are considerably less formal than UML state diagrams. As for user interface design, the level of formality may vary, both because the sub-domain is difficult to formalise, or because the participants do not have the need or ability to manage them.

The task, dialogue and concrete interaction perspectives all have correspondences to the functional/process, behavioural and data perspectives of systems engineering. Throughout the development process, the coupling that exists between the system and its interface will show up as mutual design constraints that must be identified and managed. For instance, processes provide a context for tasks, and tasks will have to be performed according to the process dynamics. Similarly, a requirement for progress feedback during database search, will translate to a requirement of the underlying implementation in terms of SQL database queries. Formal representations may play a key role, depending on the development approach chosen and the conditions it provides for using them, as discussed in Section 2.2, Chapter 2.

It seems that a fourth dimension along the user (interface) - system dimension is straightforward to integrate into the presented framework. The main motivation for doing so is to

coordinate specification and development of the user interface and the underlying back-end system. Whether a separation between user interface and system issues is meaningful conceptually or practically, depends on the application. Several reasons can be given for such a separation: 1) to isolate the user (organisation) from issues and details they have no interest in, and 2) to split the development into parts, suitable for parallel development or development by different teams. In case of 2), adding a fourth dimension may be helpful for coordinating development and to manage changes in each part to ensure consistency. This requires that the user and system planes are suitably aligned, to allow identifying representations covering the same issue. For instance, during the presentation of a medium-fidelity prototype of an expenditure application, the user may ask where the currency symbol is. This may reveal the need for handling several currencies, which in turn affects the design of the database. The relation between the window component presenting the monetary value and the database field that stores it, is represented by a path between the user and the system planes, as illustrated in Figure 14. Without knowledge of this relation, it is difficult to keep the user- and system-oriented representations of the total system consistent.



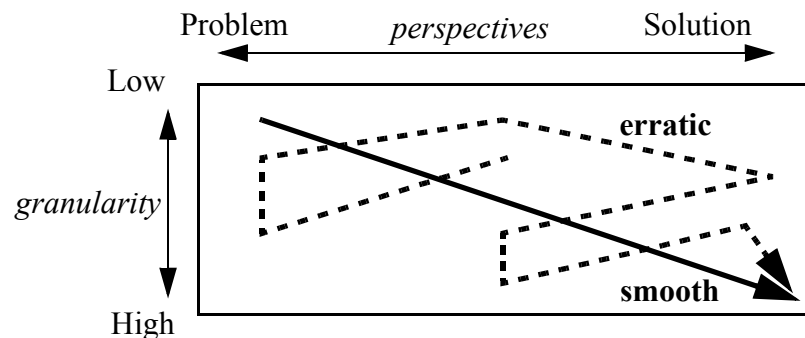
**Figure 14.** Correspondence between user and system views of the design

Moving along the user - system dimension is easiest if the same or compatible representation languages are used. In our work presented in the following chapters, we have chosen to base several of the languages on languages from the engineering tradition, partly to provide better support for moving along this dimension. With the same languages, it is easier to see how two models differ, e.g. how a conceptual system-oriented model integrates several possibly inconsistent user-oriented models for different user groups, or alternatively, how a single user's model may be a filtered and simplified version of the system model. Differences may be due to several factors, e.g. that a focus on organisational, cost or technical issues leads to different decisions, and that the system must integrate models and often resolve inconsistencies. Although we have chosen not to detail the user - system dimension, we hope to show, in the following chapters, how the user interfaces languages correspond or relate to their system-oriented cousin, whether it is the languages or their typical usage that differs.



## 3.5 The role of design representations

In Section 3.2 we noted the difference between engineering and creative designer traditions. The engineering tradition stresses the “chain of transformations from the abstract (requirements) to the concrete (resulting artifact).”, while the creative designer tradition focused more on the “tight interplay between problem setting and problem solving” [Löwgren, 1995]. Accordingly, from the software engineering point of view, it is useful to partition the domain into perspectives, e.g. to structure the process, provide better tool support and support formal analysis. The natural process to follow is indicated by the “smooth” path in Figure 15, i.e. a linear transition from abstract and high level to concrete and low level. The creative designer, on the other hand, needs integrated representations, not separation of concerns, so the idea of separate perspectives may seem artificial and constraining. The “erratic” path through the design space in Figure 15 illustrates a more “creative” design process. This difference suggests that the representation plays a different *role* in engineering design work than in creative design. Although we are focusing on user interface design representations, the roles that are identified and discussed in these sections, are relevant for design in general.



**Figure 15.** Smooth and erratic paths through the design space

The role that the engineering approach has focused on is what we may call the *semantic* role of design representations. The purpose of this role is to capture the full semantics of the domain, whether problem- or solution-oriented, low or high granularity. This essentially means covering the two dimensional space shown in Figure 11, e.g. by using one of the nine models in Table 1, Chapter 2.

No designer works on her own; different knowledge and skills are needed for both understanding the problem and deriving a suitable solution, i.e. a usable interface. To be able to discuss their understanding of the problem and its proposed solution, designers need to communicate by means of some representation medium. Later, when proceeding with the realisation of the designed interface, the implementers must be able to understand the design. A physical representation of the design is needed, to support the dialogue and asynchronous communication. This will be called the *communicative* role of design representations.

Based on an understanding of the problem (and its representation), a solution must be derived. The problem-solving process may proceed through many steps, where each step provides increased understanding, concreteness and detail, from high-level goal statements to task-descriptions, from tasks to dialogue structures, and from dialogue to interaction

details. Crucial to each step is an understanding of how the decisions at each step affect the usability of the product. Important questions are: How can different user interface elements support the user in reaching the relevant goals by performing the desired tasks? How do individual user interface elements affect the usability of the interface, and how are they composed into a usable whole? How are abstract dialogue elements mapped to concrete ones, to preserve the consistency of the platform? Clearly, a representation of both the problem and its proposed solution for each step can be useful in supporting such synthesis, both as a source of *creative inspiration* and as a *constraining guide*. As we focus on constructing a complete design from many parts, we will call this the *constructive* role of design representations.

The designer's understanding of the problem will be influenced by the current organisation of work, i.e. the current practice, and a solution to the observed problems may both be new ways of working and new tools to support work. Identifying problems with both current work practice and current (use of) tools requires analysis of the current state of affairs, as captured in the design representation. Similarly, a proposed design solution must be evaluated against the goal and task structure resulting from the task analysis and against stated usability goals and metrics. In general, the representation of the result of each design step must be analysed and compared with the requirements and constraints from previous steps. This is the *analytic* role of design representations.

The realisation of the "physical" user interface requires a representation that is understandable by the deployment platform. This normally involves encoding the interface behaviour in a suitable programming language, and transforming this into a lower-level executable representation using appropriate tools, but may also be handled using higher-level executable specification languages. Ideally it should be possible to evolve the design representation into an executable a form, with little human effort. This is the *executor* role of the design representation. Note that this role is different from the semantic role, since the executable representation need not explicitly represent the whole design, like goal and task.

Role	Objective
Semantic	Capture domain knowledge
Communicative	Communicate domain knowledge as represented
Constructive	Guide and constrain the further development of (solution oriented) representations
Analytic	Interpret and evaluate current representations
Executor	Support finalisation and execution of solution

**Table 3.** Roles of design representations

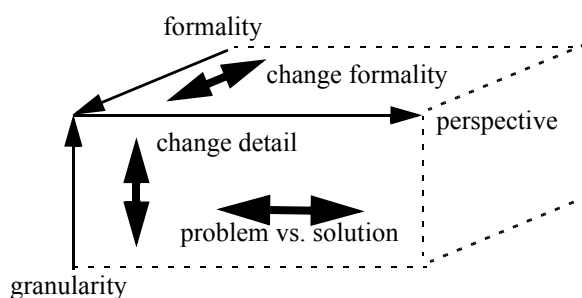
The identified roles, as summarized in Table 3, are related to different ways of *using* representations. [Wieringa, 1989] in contrast, suggest three roles of models, descriptive, normative and institutionalizing, that are related to different ways of *interpreting* representation with respect to their intention and force similar to speech acts. The idea that representations have different roles can be related to the quality framework of [Krogstie, 1995], and his notion of (aspects of) *quality*. Krogstie introduces five model qualities, physical, syntactic, semantic, pragmatic and social, defines the goal/measure of each quality and suggests how high quality models can be attained through different activities. Our semantic role corresponds to his semantic quality, i.e. the extent to which the domain is fully and correctly cov-

ered by a model, or rather, to *perceived* semantic quality, which is the semantic quality relative to a particular actor's personal understanding of the domain. The semantic role is also related to one part of (the dual) physical quality, i.e. the extent to which the designers knowledge is (faith)fully *externalized*. The communicative role is related to the other part of physical quality; the extent to which others can *internalize* the representation, as well as pragmatic quality; the extent to which others can *understand* the representation. The constructive and analytic roles are related to how the representation supports reasoning, and is similarly related to pragmatic quality, since understanding is a requirement for reasoning. The difference between constructive (synthetic) and analytic reasoning is whether the reasoning is directed towards completing the design or validating or verifying the design. Finally, the executor role is related to the pragmatic quality with respect to technical actors, i.e. the extent to which computer-based tools understand the representation.

Krogstie's discussion of modelling language quality is related to our effort at understanding the usage of design representation, where classes of representations corresponds to modelling languages. The goal is both to understand which qualities are important for various ways of using representations or models, and how (well) different classes of representations/languages can play various roles or be used for different purposes. To fill a particular role a representation may focus on a particular model quality, and similarly, a method focusing on a particular role should focus on the corresponding modelling language quality.

## 3.6 Representations and process

The absolute and relative placement of the areas of interest, tells us something about what part of the domain is covered by a particular approach, and how formal it is. In the process of designing a user interface it is expected that more than one approach and representation is needed. The representation used must be chosen according to the needs of the process and the participants, and the intended role of the representation, at any particular moment.



**Figure 16.** Movements in the representation space

The change in usage of design representations can be interpreted as movements in the three-dimensional representation space defined by our classification framework. These movements are indicated in Figure 16 by the thick arrows, and are detailed below. The engineering way of proceeding from problem to solution corresponds to a series of movements from back to front, i.e. increased formality, left to right, i.e. problem to solution and top to bottom, i.e. low to high granularity. The creative process includes movements in all six directions. This was illustrated by the path labelled “smooth” in Figure 15. For instance, a shift

of focus from creative idea generation to design consolidation may require a corresponding shift from informal to formal design representations. Validating a design together with an end-user representative, may instead require a more informal (non-executable) representation, which requires the opposite movement.

With three dimensions and two directions, we get six different basic *movements*.

### **Problem/solution dimension**

1. A solution to a specified problem is derived, through the use of some kind of design knowledge. This is the “textbook” flow of a problem solving process, first define/specify the problem, then solve it according to the constraints provided by the problem specification. The representation should support looking “forward” towards constructing one or several solutions, i.e. play a *constructive* role. Since this movement represents a shift in focus or perspective, the support given by the modelling language(s) alone may be limited.
2. The problem that a solution solves is derived. This can be thought of as reconstructing the explicit constraints that a solution satisfies, and may be regarded as reengineering a specification, or analysing a solution for the purpose of comparing it against a previously formulated specification. The representation should support looking “backward”, i.e. play an *analytic* role.

### **Granularity dimension**

3. From a high-level description details are constructed. This is similar to movement 1, in that it requires satisfying the constraints given by a more abstract specification. Accordingly, the representation should play a *constructive* role. However, it will normally be easier to match a description against some constraints within than across perspectives.
4. A higher level description is derived from a lower level one, i.e. the aggregated characteristics are constructed from an analysis of one level, e.g. to compare against the higher level specification. Similar to movement 2, the representation should play an *analytic* role. This depends on the compositionality/constructivity of the language used, i.e. the ability to derive aggregated characteristics of the whole from the characteristics of the parts and their configuration.

### **Formality dimension**

5. A representation of some problem or solution is made more formal. As for movements 1 and 3, this is the “normal” movement, in that vague ideas and knowledge are transformed into less ambiguous and more precise descriptions. To ensure proper formalisation, it is important that the meaning and common understanding present in the context of the process, be made explicit through communication among participants, hence the representation should play a *communicative* role.
6. A formal representation is given a less formal form. This may seem like an unnatural movement, but may make sense in a context where participants are unfamiliar with formal notations. It will be important to create a context for giving meaning to

the less formal representation, based on the original, more semantically rich one. The representation should play a *semantic* role, since the meaning must be clear before being transferred to a meaning-giving context for the participants.

The last dimension and corresponding movements deserve additional comments. Proponents of the model-based approach typically describe their method as moving from top to bottom and left to right in the formal plane, i.e. high-level models are decomposed into lower level ones, and the problem is specified at one level before a solution for the same level is derived. This is a naive view of what actually happens during problem solving and design. As discussed in Section 3.2, the creative designer approach makes active use of the solution media/material in the search of a fruitful understanding of the problem and solution. Concrete representations are used instead of abstract ones, and hence, informal descriptions like sketches are favoured since they stimulate the creativity better than abstract models. Later, these must be made more formal to be useful in the further development.

In [Nonaka, 1998], a model of *organizational knowledge creation* is presented. Based on the distinction between *tacit* and *explicit* knowledge, four modes of *knowledge transfer* that the knowledge management process should support, are described. The distinction between explicit and tacit knowledge is closely linked to the relationship between formal and informal representations. Explicit knowledge is knowledge that has or can be codified such that it is available to others, given that they understand the “code”. The code can be natural language or any formal or semi-formal notation. The more formal the representation, the more objective, context-independent and explicit the knowledge will be. The main point is that the representation is external to the originator, independent of context and can be stored and transmitted (to others). Tacit knowledge, on the other hand, is personal, context-specific and subjective and cannot be stored or transmitted directly to others. Tacit knowledge is made explicit through a process of *externalization*, while explicit knowledge is made tacit through a process of *internalization*. The latter is often needed for actually using knowledge creatively, instead of mechanically. According to this theory, any problem solving process implies knowledge creation and hence, there must be a constant conversion of knowledge between the tacit and explicit forms, as shown in Figure 17 (from Nonaka, 1998).

As mentioned earlier, the role of the representation is both to capture knowledge and to support the design process, which in the context of this knowledge creation model means the four transfer modes:

- *Socialization* - Tacit knowledge is shared with others, through dialogue. The main purpose of the representation is not capture of knowledge in explicit form, but rather support of social interaction, with the goal of getting familiar with the design context and with design alternatives, analogous to the language game of design. Informal representations like scribbings and sketches are best suited, since they support story-telling, annotations and physical

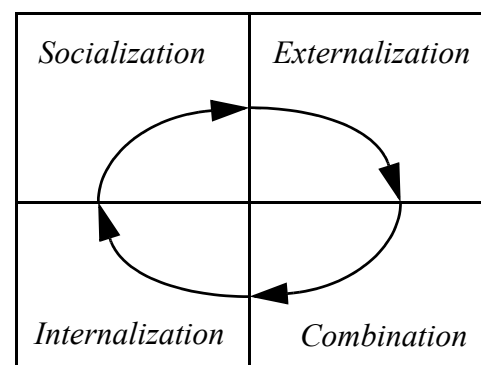


Figure 17. Nonaka's modes of knowledge transfer

interaction. The process (people, time, place) provides the context for giving meaning to the representation, without which the informal representation has no meaning. After the transfer, the meaning is embodied in the process' participants.

- *Externalization* - Tacit knowledge is made explicit, by verbalizing in a dialogue among designers. Again the representation should support this dialogue, since externalisation often is a social process, although it could take the form of an "inner dialogue". Throughout the process, the representation will shift from informal to formal, e.g. from sketches to screen definitions or diagrams. The authoring process may be supported by tools for checking or increasing the representation's syntactic and semantic quality.
- *Combination* - New explicit knowledge is created by linking existing explicit knowledge. This process depends on representations that supports analysis, whether by humans or machines, i.e. the main advantage of formal models. Different perspectives may be combined, or representations of various levels of detail. The reader's understanding of the representation language is important, and a various tools/aids may be used to increase the pragmatic quality of the representation.
- *Internalization* - Tacit knowledge is created by acting upon explicit knowledge in a process of "learning-by-doing". Again the process of reading the formal representation is important. However, the learning environment, e.g. prototyping or simulation tools, is equally important, since the tacit knowledge requires a personal experience with sensational and emotional engagement.

Throughout the design process there will be a drift towards more explicit and formal representations, as the design is committed to and the need for transfer to "outsiders" increases. Subsequent analysis and evaluation corresponds to "combination" in that elements of design are considered as a whole and new characteristics are derived, e.g. to judge complexity, learnability and usability. The last transfer mode can be compared to validation by means of (usability) testing of prototypes, since it means experiencing the specification in a more physical way.

When using informal user interface design representations the intended meaning will depend on the design context. The process of formalising the representation requires interpreting the informal representation. If the original design context is not available, e.g. is forgotten, it may not be clear how to disambiguate the informal representation. Suppose a design document includes a labelled text field with a date in it followed by a "Change" button, a probable interpretation may be given within several of the main perspectives discussed above:

- **task/domain**: that the user needs to change a date at some point of a task or that the object attribute named by the text label will refer to a date
- **dialogue**: that at some point the user will need to interact with a date field and that the button is a means of changing an object attribute
- **concrete interaction**: that this particular date attribute should be input using a text field, that this date field should be laid out according to this particular position and dimension or that a button should be used instead or in addition to a menu item for activating the change (?) function.

These are not mutually exclusive interpretations, the sketch may in fact imply all of them. In general, to make it possible for others to interpret the informal representation correctly, the design team must either include others in a suitable context for understanding their design representations, or remove the need for establishing this context by making more explicit and formal representations.

## 3.7 Formal representations revisited

Although the user interface design community has been reluctant to use formal representations for design, work within the field of model-based user interface design shows that formal representations can be valuable, also from a “pure” user interface design perspective [Szekely, 1996]. Among others, models of tasks, objects and dialogue can support both the design and evaluation of user interfaces, based on several important characteristics of formal representations:

- The meaning is well-defined and independent of the context, i.e. user, time, place.
- Important characteristics can be derived or checked, like consistency and completeness.
- Different parts and aspects of a system can be compared and analysed as a whole

What is actually considered formal (enough) varies, e.g. a critic of UML is its lack of formality [Bergner, 1998], while from the perspective of user interface designers, UML is very formal indeed. The characteristics above require a mathematical foundation, e.g. algebra or logic, which most of UML lacks, but which is supported by many existing formalisms for information systems engineering and user interface design. It is instructive to look at the domain of structural engineering and some of the properties we consider crucial for its success with utilising formal models:

1. A limited number of characteristics are needed to describe the basic elements of the construction, as well as compositions of these elements. For a bridge this includes weight, mass distribution and strength.
2. These characteristics may be computed for a composite construction, based on the elements and how they are composed. Weight is a simple sum of the parts, while mass distribution is easily calculated from each part’s mass and position.
3. Requirements for the construction could be formulated in terms of these characteristics. The forces the bridge must withstand may e.g. be translated to a required strength.

In short, these properties ensure that the domain remains conceptually simple, even though any specific construction may be very complex and require advanced methods of design. For practical purposes it is important that the mathematics used for computing and analysing the characteristics is well understood and practical to use for any reasonably sized problem. This is true for structural engineering, e.g. by using the Finite Elements Method to partition the computational problem.

Within the engineering approach to systems development, the terms *compositionality* and *constructivity* have been used for characterising theories which “provides rules for reasoning on compositions of specification modules without knowledge or anticipation of the implementations of these modules” [Gaudel, 1994]. This means that the characteristics of a system only depends on the components’ characteristics and how they are composed, similar to item 2) in the list above. The concept of compositionality originated in the philosophy of logic and language and referred to the property that the meaning of a statement was the “sum of its parts”, i.e. could be derived by looking at the parts and their composition only, while ignoring the context of the statement or interpretation and the interpretand [Dict-Mind@]. Natural language does not in general have this property, but formal languages in general do.<sup>1</sup> The original concept of compositionality makes it possible to understand parts of a model of a domain, in the form of a set of formal statements, without the need of considering how other parts may affect it. In other words, adding a statement would not change the meaning of existing statements, although it may of course change what can be derived from the whole, e.g. by making it inconsistent.

Compositionality alone does not, however, guarantee that the aggregated meaning of a set of statements can be derived in practice. For instance, there may not be a limited set of characteristics for defining the meaning, nor a practical mathematical theory for deriving characteristics. However, when compositionality is used in modelling literature, the former seems to be implied and the latter at least hoped for. Constructivity seems to be given a similar meaning, i.e. a limited set of characteristics are used both for specifying desired properties and describing elements of the domain, to be verified against the specification. For instance, the port interface of a process and conditions for flow of data, might be derived for any level of a process hierarchy and compared to the specified requirements. Hence, the advantages of compositionality, seems to require both:

1. *conceptual abstractions* of a domain according to the three properties listed above
2. *practical mathematical theory* on which the concepts are based

The latter requirement presents several obstacles, e.g. it requires precise, complete and consistent models. In addition, the current mathematical theories are intractable for realistic problems, so that only parts of a model may be fully analysed. While this may seem disappointing, the conceptual abstractions alone may provide advantages as an analytical tool for the developer, even without using the power of (computer-based) mathematical analysis. This is the stand we take in this work: Models based on conceptual abstractions of the problem and user interface domain are useful tools for developers, given that they support the necessary roles in the development process, as identified above.

---

1. The former is due to cultural and social conventions, while the latter is by design.



# Chapter 4

## Task modelling

Task modelling can be viewed as a way of formalising the results of a task analysis, the goal of which is to understand and capture the nature and structure of user population, their environment, goals and work activities. When used for evaluating existing user interfaces, the task model should and will be highly dependent on the existing user interface. If, on the other hand, the task model is focused on design, it should try to remove the constraints embodied in the existing user interface by avoiding reference to the artefact. Hence, in the former case, the task modelling will be more solution-oriented, while in the latter case, it will be more problem-oriented. It is the latter usage that is the target of our work, i.e. task modelling used for designing a new user interface, rather than evaluating an existing one.

### 4.1 Introduction

Task models were in Chapter 3, “Design and representation”, classified as a medium-level, formal design representation, with a problem-oriented perspective. According to the same classification, workflow models were considered a higher-level representation within the same perspective, whereas dialogue models were similarly levelled, but more solution-oriented. Hence, one may view a task model as being in-between workflow and dialogue models, in the sense that a workflow model will provide a context for task modelling, and the task model a context and a set of constraints for the dialogue model. In a top-down process, a workflow model will play a *constructive* role with respect to a task model, since a task model will have to be constructed as a decomposition of the workflow model. Relative to the workflow model, the task model will play an *analytic* role, since it must be interpreted and evaluated with respect to the workflow context. For both roles, it will be an advantage if the respective modelling languages are based on similar concepts and constructs, while still being adapted to the appropriate level within their common perspective. The relation between task and dialogue models is similar, in that they play similar constructive and analytic roles with respect to each other. However, since they represent different perspectives, it is not desirable to base a task modelling language on a dialogue modelling language and hence, more difficult to relate the models.

In this chapter, we therefore take a look at task modelling in the context of workflow modelling and suggest a task modelling language based on interpreting a workflow language within the task modelling tradition.<sup>1</sup> We first motivate integration of workflow and task

modelling and then present and compare workflow concepts with those used in task modelling. The workflow concepts are shown to have task modelling correspondences, and possible benefits of integrating them are suggested. We show how a workflow model can be utilized when modelling the tasks of a workflow participant and illustrate how end-users can benefit from task enactment as a practical result of workflow and task integration. Our task modelling language TaskMODL is subsequently presented, and in Chapter 5, “Dialogue modelling”, the relation to the more solution-oriented dialogue model will be discussed.

## 4.2 Workflow and task modelling

Understanding the nature and structure of work is crucial for building successful information systems (IS) for organisations, for at least two reasons. First, for an IS to be understandable for the organization it is part of, it must take current work practices into account. Second, because an IS to a certain degree determines both *what* work can be done and *how* the work can be performed, it must be designed according to the organization’s objectives and goals. An organization’s IS must support work being performed at three levels, the organizational, group and individual levels. This notion of level corresponds to the granularity dimension of the classification framework introduced in Chapter 3. Work performed at the organizational and group level is often described using *workflow* modelling languages. Workflow models are useful for IS development, both as a tool for capturing and formalizing knowledge about current and desired work practices, and for the design of the computerized information system (CIS). For describing small-team and individual work task models are used, based on a *task analysis*.

As argued in [Traunmüller, 1997], we need to integrate the various levels to actually build useful software for supporting work, i.e. bridging the gap between organisational, group and individual work. In the following sections, we look at integration of workflow and task modelling *language* and *models*. The main goal is to study how workflow concepts can be useful for task modelling:

- At the *language* level, workflow modelling concepts can be used for task modelling.
- At the *model* level, making the interaction between individual and group work explicit can help us make task models that take the work context into account.
- At the *enactment/performance* level, having an integrated approach to workspace design and user interface execution may result in better human workplaces, where the values and foci of the workflow and task based approaches are balanced.

Section 4.3 will introduce important aspects of work, as identified by the workflow community and found in workflow modelling languages. The aspects will be interpreted in the context of task modelling and integrating these into task modelling languages will be discussed. Section 4.4 will show how workflow models might influence task modelling, and suggest

---

1. Part of this work has previously been published in a shortened version in [Trætteberg, 1999].

how the theory of speech acts might be used in user interface design. As a conclusion and direction for further work, we in Section 4.5 turn to how an integrated machinery for workflow enactment and user interface execution might benefit the end user. Throughout the paper, we will use the Action Port Model (APM) [Carlsen, 1998] workflow modelling language and a workflow example, both to illustrate the concepts and show how workflow models can aid the design of user interfaces for workflow participants.

## 4.3 Aspects of work and its description

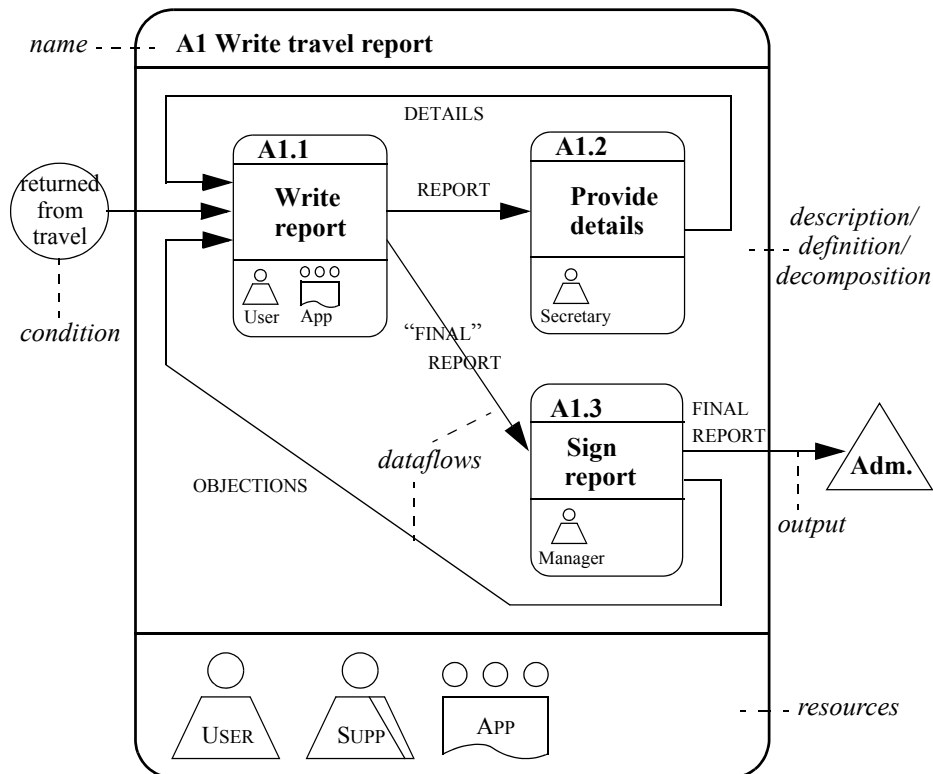
Work can be described along several dimensions, your choice of dimensions depending on your goal. In traditional workflow modelling, the goal has been threefold:

1. process understanding, e.g. as a basis for business process reengineering (BPR)
2. workflow management, including work allocation and monitoring
3. work support and enactment, including workspace design, client support.

[Marshak, 1997] identifies four dimensions for describing work: the *action structure*, the *actors* that perform the work, the *tools* that are used and the *information* (sources) that are needed. To illustrate these workflow dimensions and the potential use for task modelling, we first take a look at a workflow example and its APM model, as shown in Figure 18. A person (USER) must write a report upon return from a travel. He has to his aid a group of persons (SUPP) and we would like to further support him by designing a piece of software (APP). These three entities are modelled as *resources* of the top-level WRITE TRAVEL REPORT task, by including them in the resource bar. The action structure is based on the decomposition of how USER works on the report and how he interacts with the SECRETARY and MANAGER to fill in details and gather and respond to objections. This workflow model will be used as a starting point for a task model for the APP software that USER uses to perform the main task and interact with the others. Now we will present the workflow concepts, detail the notation and discuss the relation to task modeling.

The most fundamental concept is the *action*, depicted in Figure 18 as a rounded rectangle, although different names like task and act are widely used. Actions are the causes of all change and constitute the small steps leading to the goal. For an action to be possible, certain implicit and explicit pre-conditions must be satisfied, e.g. the action should be relevant for the goal and necessary resources like information and tools must be available. The pre-conditions of an action may depend on other actions, making it natural to view all the actions as a structure of dependencies.

The dependency structure of workflows is usually based on dataflow and/or speech acts [Searle, 1985]. The former defines *necessary* constraints on action sequences by specifying what input data are required. In the example, the manager must receive the report to be able to sign it or return objections. Explicit pre-conditions or control flow can also be included, as in the example were the return from the travel triggers the workflow. In the speech act approach, action sequences are defined on the basis of institutionalized dialogue patterns, whether formal (legal) or informal (social). This normally gives more *restrictive* pre-conditions.



The A1 decomposition defines how writing a travel report is performed. As can be seen, the definition of an *APM* action consists of four parts: the identifier and name, the definition of the action as a decomposition, the driving information as input and output, and the required resources like actors and tools.

**Figure 18.** The APM example model.

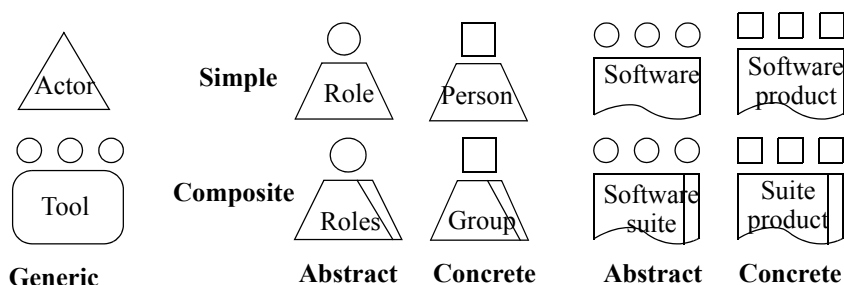
tions than the dataflow approach. For instance, the secretary may be required to always check the travel report even if no details are missing.

The full semantics of an action is rarely formalized in workflow modelling languages, since it may be far too complex. However, support for describing the action dependency structure is normally strong. To reduce complexity the structure can be made hierarchical by defining composite actions, like A1 in the example. Such actions are often denoted by a distinct term, e.g. “job”, “process” or “activity”, highlighting that these are complex and longer lasting actions. Composite actions can be abstract, by defining the external characteristics and leaving the internal structure unspecified until needed.

The action concept corresponds to the task concept, and the action hierarchy to the task/sub-task structure, found in task modelling languages. At the model level, we believe low-level workflow actions may correspond to high-level tasks. Several task modelling languages are based on dataflow, e.g. the Activity Chaining Graphs used by TRIDENT [Bodart, 1995], giving action dependencies similar to some workflow languages. However, most task modelling languages are based on explicit sequencing primitives like ordered/unordered sequence, repeat and choice for defining action sequences. The primitives are used in two ways, similarly to speech act based dialogue patterns: 1) prescriptively, by specifying/limiting how actions must/can be performed, or 2) descriptively, by describing how actions are performed in practice. In such a task model it is not explicitly expressed whether a particular sequence is due to a necessary pre-condition or simply is socially conditioned.

*Actors* are the beings that intentionally perform actions. Actors have a (mental) state guiding their behaviour, e.g. a purpose or goal state to reach, responsibilities and commitments. Actors both perform actions and decide which actions to take. In the example, two kinds of actors of A1 are externally visible, while three are defined internally (see notation in Figure 19). Technically, actors can be viewed as resources with certain characteristics and abilities that are required by the actions. By listing the needed characteristics, like rights and privileges, knowledge and abilities, actions implicitly define who can perform them.

As with actions, actors can be composed, giving a *group*. While groups usually match formal organization structures, they can be defined to contain arbitrary individuals. Although it is not always meaningful to say a group performs a particular action, it may be to say that an unspecified and able member of a group will perform it. Alternatively, a whole group may take part in a composite action, i.e. the group is the aggregate of the performers of the composite action's subactions. In the example, the composite SUPP actor models the group of individuals supporting the USER actor.



Both type of symbols have a generic variant, as well as specific symbols for simple/composite and abstract/concrete actors. The role is a simple and abstract actor, while specific persons are simple and concrete.

**Figure 19.** The APM actor and tool symbols

While groups are defined extensionally, by listing their members, *roles* define sets of actors intentionally, by listing their common characteristics. The characteristics are typically those that are required by actions, i.e. roles are used to define/summarize sets of necessary requirements of actions, like authority, knowledge and capability. Roles can be treated as (composite) resources required by actions, that must be filled by actors before allowing them to perform an action. Actions are then said to be performed by these roles. A particular actor can play several roles, although not necessarily for all actions or in all contexts.

The most natural task modelling interpretation of the actor and group concepts are users and user groups. Users and user groups represent concrete individuals, and are referred to when describing how tasks are actually performed (by this group), i.e. the current work practice. Roles correspond naturally to user stereotypes, i.e. particular kinds of typical users, since they refer to characteristics that some but not necessarily any individuals might have. Stereotypes are typically introduced when there is no useful definition of a single “generic user”, so instead several definitions are used.

*Objects* are the material, mental or representational things constituting the domain that actions are performed within. Objects are used both when representing the information that is acted on or upon and for modelling contextual information used in actions. The term ‘object’ is used informally, since the formalism used for describing the domain is not neces-

sarily object-oriented. In the example, the REPORT could be a text document, the DETAILS a spreadsheet and the OBJECTIONS a natural language email. Objects are created, destroyed and modified by actions, whose semantics may be described in terms of objects' state, e.g. by pre- and post-conditions. The objects granularity can range from whole databases and documents to records and words. The detail of the object model may similarly vary, depending on the goal of the model.

Most task models assume there is a model of the domain or the application data, that tasks use or operate on. The tasks' semantics may be fully defined in terms of these objects, e.g. using algebraic equations, or just refer to the main domain concepts. As for workflow, the level of formality and detail may vary, although it is common to use object-oriented or similar models with concepts, relations and attributes. We expect workflow objects, if further detailed to a relevant level, to be directly usable in task models.

A *tool* is a special case of the general term *resource*, which has been used for every prerequisite for performing actions, including the actors performing the them, the objects acted on and additional contextual information needed. At the granularity of workflow, tools are typically applications or components in integrated systems. A tool can be concrete or abstract, where the former names a specific application, e.g. 'Eudora', and the latter refers to an application class, like 'email client'. In addition, tools can be composite by listing several applications (classes) or components in a suite or referring to the suite as a whole, e.g. 'Internet access tools' or 'office package'.

The enactment engine of a workflow system has the task of bringing the work(flow) to the worker, by providing the relevant information and activating the relevant tools for an action. Extending this principle to user interface tasks requires a suitable interpretation of the tool concept in the task modelling context. Tools are software elements supporting the performance of actions, which at the granularity of user tasks correspond to *dialogue elements*. The tool concept provides the link from the (problem-oriented) *specification* of what support the user interface should provide, to the user interface (solution-oriented) *design* elements actually providing it. The idea of *task enactment* will be further discussed in Section 4.5.

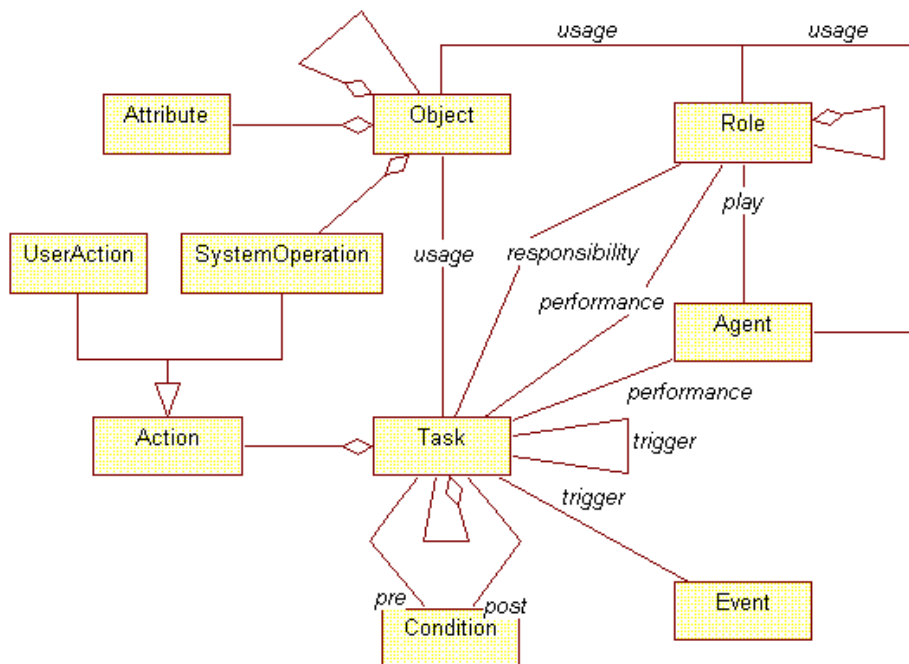
Workflow concept			Task interpretation
Generic	Abstract	Composite	
<b>Action:</b> basic unit of work, data flow or speech act based	Delayed definition, action template parameter	Provides hierarchical composition, also called process, job, activity	Task, task/subtask hierarchy <ul style="list-style-type: none"> <li>• Data availability defines necessary pre-conditions</li> <li>• Work practice as additional constraints</li> </ul>
<b>Actor:</b> intentional beings performing actions	<b>Role:</b> Intentionally defined set of actors, often based on actor characteristics	<b>Group:</b> Extensionally defined set of actors	User, user group, stereotype <ul style="list-style-type: none"> <li>• Multi-user task models</li> <li>• Multiple work practices</li> <li>• Design targeted at different user groups</li> </ul>

**Table 4.** Workflow and task modeling concepts

Workflow concept			Task interpretation
Generic	Abstract	Composite	
<p><b>Object:</b> part of (representation of) domain</p> <p><b>Tool:</b> software needed for performing actions</p>	Application class	<p>Part-of hierarchy</p> <p>Application suite, integrated components</p>	<p>Object</p> <ul style="list-style-type: none"> <li>• Operated upon</li> <li>• Contextual information</li> </ul> <p>Dialogue element</p> <ul style="list-style-type: none"> <li>• Link between task description and interface design</li> <li>• Task based enactment</li> <li>• Component composition</li> </ul>

**Table 4.** Workflow and task modeling concepts

Table 4 summarizes our analysis of Marshak’s four dimensions of workflow, their correspondences with and opportunities for task modelling. The (re)interpretation of Marshak’s four dimensions in a task modelling context, suggests that workflow concepts have natural correspondences in task modelling languages. For further evidence we take a look at the ontology or meta-model for task modelling presented in [van Welie, 1998] and shown in Figure 20 as a UML class diagram. The ontology is an attempt at clarifying the concepts and relationships of existing task modelling frameworks.



**Figure 20.** Ontology or meta-model for task modelling languages [van Welie, 1998]

The main concept and structure is the task hierarchy, analogous to our action structure. Various kinds of task sequencing is provided by the ‘Task uses Object’, ‘Event/Task triggers Task’ and ‘Task pre/post Condition’ relations, covering both dataflow, data triggered and explicit sequencing. Our actor concept is covered by their agent and role concepts, although there are some differences. Most notably, agents can be software components, and roles have an explicit relation to the task they (can) perform, instead of implicitly by listing the resources that tasks require. The object model is simple, allowing for attributes, actions and a hierarchical is-a and containment structure. The task semantics is partly defined in terms

of objects; they are triggered by attribute changes, can perform attribute changes and provide post-conditions referring to objects' states.

When designing multi-user applications there is a need for introducing concepts for describing cooperation and interaction between tasks. ConcurTaskTrees [Paterno, 1997] targets applications for small groups where cooperation is required. The main idea is allowing explicit cooperation and communication tasks, which span the task hierarchies for individuals. In addition to the usual task hierarchies for the individuals' tasks, a hierarchy for the cooperation and communication tasks is modelled, that refers to the individual task hierarchies. The modelling concepts are consistent with the ontology in Figure 20, as well as our idea of moving up from individual work to multi-user group and group work, by adding levels of abstraction.

The main difference between the task and workflow approach is this task ontology's lack of a tool concept, i.e. the software artifact supporting the user when performing tasks. Although we have system operations, these are part of the domain model and not the user interface is to be designed. The difference is due to the focus of each approach: task analysis and modelling focuses on what the user does or should do and not by which means it is done. It is considered important to separate the act of understanding goal/action structures and designing for its support. Workflow modelling on the other hand has a strong focus on bringing to the user the tools necessary for performing the required actions. Although we agree that separating "what" from "how" is important, we believe including the concept of design objects could aid the transition to design. Existing approaches have included a tool concept as a way of integrating task and dialogue models in a fully task and model based approach to interface design, e.g. MOBI-D and MOBILE ([Puerta, 1997] and [Puerta, 1999]).

Workflow modelling's focus on active computer support, has lead to the introduction of concepts for more dynamic flow of work than the ones presented above, including *routing* and *rules*. The performance of specific actions is normally initiated by data flow or explicit control directives, which will determine the possible action sequences. There may be good reasons for delaying some decisions with respect to information and control flow. First, the actual performing actor may not be decided until the latest possible time during enactment. Second, each potential actor may have a different preferred way of performing actions. Third, there may exist several versions of the same abstract action, and which concrete action to actually perform may best be determined based on the actual information flow content or other contextual information. In a workflow model, *routing* defines which tasks that will receive and maybe be triggered by information, the actual information content and who will perform the task. *Rules* are used to specify how choices are made, i.e. in what situations the various routes are taken. In a task modelling context these choices are similar to *method selection* (the M in GOMS) [Card, 1980] that actors perform. Such selection occurs when there are several action sequences leading to the same goal, with important qualitative differences concerning speed and effort. Expressing the reasons for choices in rules should increase the utility of the task model. In workflow, routing and rules can also be used for selecting the appropriate tool for a task. Based on our interpretation of tools as interface dialogue elements, this naturally corresponds to dynamic selection of dialogue elements, e.g. automatic mapping from abstract to concrete elements. There already exists systems that choose concrete elements during runtime, based on space requirements, e.g. Mastermind [Szekely, 1995]. Selecting the abstract and concrete interaction objects based on the



dynamic task context seems like a natural consequence of applying the routing and rules concepts to task modelling and enactment.

## 4.4 Workflow and task models

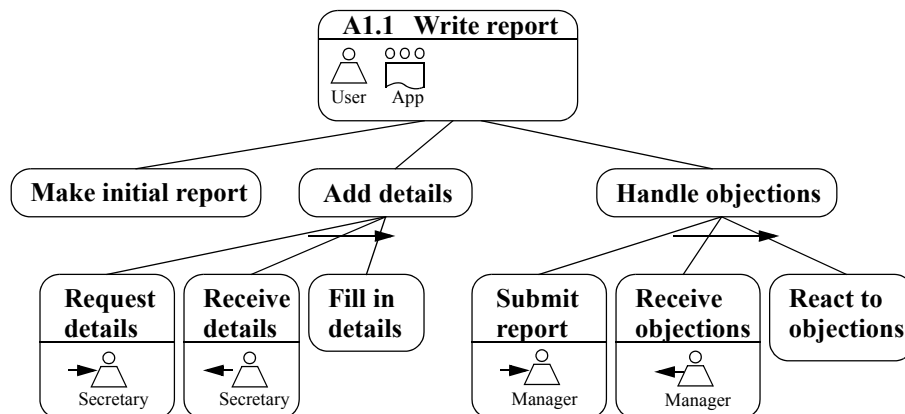
We have shown that workflow and task modelling languages have much in common, and based on the classification framework from Chapter 3 it seems worthwhile to consider integrating actual models, in addition to the modelling languages. Such vertical integration is based on the idea that workflow and task models essentially describe the same domain, but at different abstraction levels. Hence, when designing user interfaces for participants of a particular workflow, e.g. USER in the A1 workflow in Figure 18, it makes sense to use the workflow model as a context for the task model, i.e. the workflow model example. The advantage should be threefold:

1. The relevance of the task structure should be ensured, since it is motivated by goals shared by the organization.
2. In addition to using the low-level actions as the top-level tasks, most of the elements of the task model, like roles, information and tools are already identified.
3. Special workflow concepts, like speech acts and exceptions, may be given special treatment in the decomposition, e.g. by using special task structures.

Vertical integration can be done from two views, the whole workflow and the individual workflow participants. The former concerns building collaborative environments for groups, where the focus is on providing tools for communication and coordination, as well as the actual work. While such workspace design certainly is interesting, we are currently more concerned with the individual participant view. Since workflow models are system oriented, care must be taken when moving to the user-oriented perspective of individual workflow participants. Several roles of the workflow may be played by single users, so all of the tasks performed by these roles should be taken as a starting point for task modelling. This means that processes that are far apart in the high level structure may have to be tightly coupled in the task model, because the same individuals participate in them. The mapping from system to user views of the workflow is essential for task analysis, and both the task modelling process and task modelling tools must support it.

In our example, USER performs one low-level action, A1.1, which can be used as the top-level task. The three ways of starting and resuming this task, based on the single control flow and two dataflows, are defined as sub-tasks, as shown in Figure 21. The latter two are decomposed into tasks for sending, receiving and handling the relevant information. Further decomposition of information handling tasks, will depend on how the various kinds of information are acted upon, and hence, requires a decomposition of the REPORT, DETAILS and OBJECTIONS “objects”.

We see that the workflow model can provide both initial task model elements and opportunities for further refinement. The tasks that require communication suggests that speech act patterns like Winograd and Flores’ “Communication for Action” [Winograd, 1986] might be relevant to include, together with specific communication tools.



**Figure 21.** Task model for USER based on the workflow model example

A different synergy between workflow and task models is that of reusing workflow model patterns in task models. Pattern reuse is based on the idea that *human-problem-human* interaction patterns, embodied in workflow models, may also be relevant for *human-problem-machine* interaction. If you view the machine as a problem solving partner, it may be natural to structure the man-machine dialogue similarly to human-human dialogue patterns, like the ones found by speech act research. Indeed, the installation wizards found in desktop applications, seem to use the preparation-negotiation-performance-acceptance cycle that e.g. Action Workflow [Medina-Mora, 1992] is based on.

## 4.5 Towards a task modelling language

We have shown that Marshak's four dimensions [Marshak, 1997] for describing workflow, i.e. *action structure*, *actors*, *tools* and *information*, have direct correspondences in the task modelling domain. Suggestions for how these can be used in task modelling have been presented, and we have shown how a workflow model can be used as a context and starting point for a task model for a workflow participant. As the APM modelling language is based on the above mentioned workflow concepts, it should provide a good starting point for a task modelling language, given that existing task modelling *concepts* and *practices* are covered.

Task modelling is a the process of representing and often formalizing the findings of the task analysis activity. The exact information that is captured by a task modelling language heavily depends on the goal of the overall process [van Welie, 1998]. Traditional notations like hierarchical task analysis (HTA) focus on the relation between tasks and goals and hierarchical task decomposition. The GOMS family of models [Card, 1980] have a strong focus on evaluation of task performance and are deeper and more detailed. More recent task modelling languages like ConcurTaskTrees [Paterno, 2000b] are designed with computer supported design of user interfaces in mind, and hence are more formal and have tighter links to user interface dialogue architectures and languages.

We are firmly placed in the latter tradition, and have particular interest in utilising task models in the user interface design process and the overall systems engineering process, including workflow modelling as discussed above. Consequently, we need a strong coupling with

other systems engineering models, support for reuse of task models and an explicit link to user interface dialogue models. In designing the language we want to:

1. narrow the gap between workflow and task modelling, by integrating modelling concepts, while retaining the flavour of task modelling that developers are used to,
2. provide a flexible notation for representing tasks, without requiring completeness in breadth or depth, and
3. support the transition between task and dialogue modelling, by making it possible to relate constructs and models between perspectives

The second point requires further comment, since task models are used for several different purposes. Task models were initially focused on describing how tasks are performed with the current tools, so their effectiveness and efficiency could be evaluated. The basis should be observations of actual usage, which were then generalised into a common task model. We believe it is useful to capture the individual instances in a model, to make it easier to perform the needed generalisation. After task based design was introduced, as an extension of model based design, task models have additionally been used for representing how tasks will be performed with the new and/or improved tools, often called the designed or envisioned task model. The difference between the current and envisioned task models, is pragmatic rather than semantic. The current task model will have the current tool as its context, and many smaller tasks in the model will be an artifact of the current design, rather than user/problem-oriented goals. When using task models for design, such artificial tasks should first be removed altogether, for later to be reintroduced in the context of the new design, where the corresponding design model serves as a pre-condition for the task. Task models can be quite complex and it is often necessary to focus a task model on a specific (context of) use, e.g. a scenario more suitable for user validation. Such a model will be the design equivalent of user observations, the tasks performed by a particular user (type) in a certain context. These four different task models are summarized in the table below, with a suggested correspondence with Nonaka's knowledge creation process [Nonaka, 1998].

<b>Tasks models</b>	<b>User/usage specific</b>	<b>Generic</b>
<b>Current</b>	Observations of usage (Interpretation of the domain through social interaction and dialogue)	Current work practice or task (Creation of generic knowledge by means of explicit representation)
<b>Future</b>	Usage scenarios (Interpretation and validation of new knowledge in usage context)	Envisioned tasks (New knowledge is created, by combining current task and design knowledge)

**Table 5.** Four kinds of task knowledge

The problem-oriented perspective is usually split into a domain and a task modelling language. Although the task modelling language needs to be tightly coupled with the domain modelling language, surprisingly few modelling approaches provide an integrated visual notation for both. Our modelling languages are coupled in two ways:

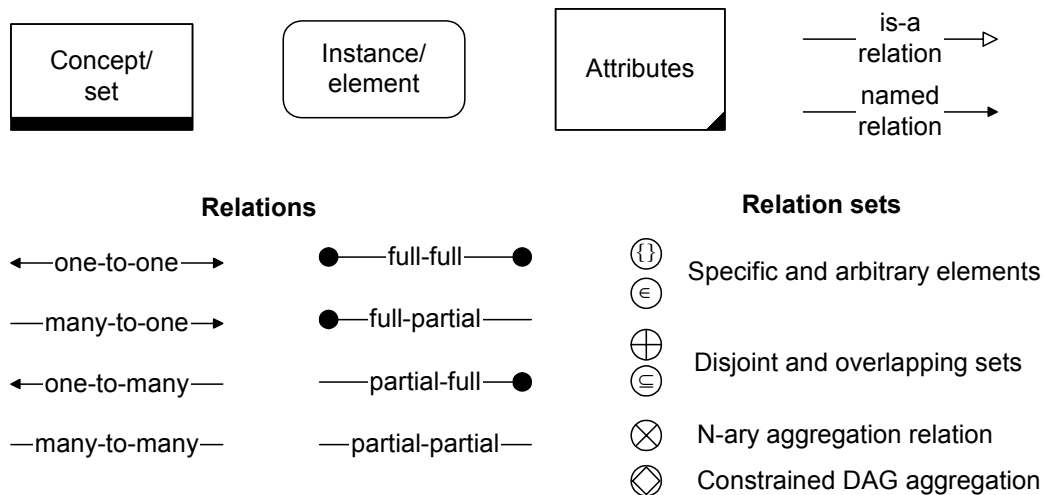
1. A task model defines the tasks the user wants or needs to perform within an application domain. The domain will itself limit the possible tasks, and although the task semantics need not be defined fully in terms of the domain, support for pre- and post-conditions will require the ability to reference the domain model.
2. The concepts found in TaskMODL can themselves be described in the domain modelling language. I.e. the domain modelling language is used for expressing the meta-model of TaskMODL.

In the following section, our domain modelling language RML is presented, with a focus on the former usage. Our task modelling language TaskMODL is designed as a blend of the above mentioned APM workflow language and traditional task modelling languages, with the goal of supporting the four kinds of task knowledge shown in Table 5. TaskMODL is described in Section 4.7.

## 4.6 The domain modelling language: RML

In this section we detail our Referent Modelling Language, which is used to model the work domain, together with the task modelling language. RML will also be used with the dialogue modelling languages presented in the next chapter and for modelling aspects of concrete interaction. We have chosen to design a new language instead of adopting UML, since we find UML too informal, design-oriented and complex (both meta-model and notation). Since UML for many purposes will be a more realistic modelling language candidate in industry, we will nevertheless compare and contrast the two throughout this presentation.

### 4.6.1 Concepts and instances



**Figure 22.** Conceptual modelling constructs and notation

The Referent Modelling Language (RML), is based on language and set theory. The basic assumption is that the real world is populated by *instances* which we observe. These instances have *identity*, and to be able to speak about them and refer to them we use *symbols* that refer to their real-world counterpart called *referents*, hence the language name. By

introducing *concepts* like persons, animals and fruit we are able to refer to *sets* of instances having things in common, by defining the characteristics that distinguish them from others. When an instance is classified as being of a particular concept, it implies that it shares these common characteristics, and the fact that these characteristics are shared, is the main reason for defining the concept. The set of distinguishing characteristics is called the concept's *intention*, while the set of instances that exhibit these characteristics is the concept's *extension*.

These two basic constructs of RML, concept and instance, is shown in Figure 23. In the example in Figure 23, we see four concepts, 'Person', 'Man', 'Woman' and 'Limb', and two instances, "Hallvard" and "a woman". In the example, every person is defined as having two attributes, 'Name' and 'Birthdate'. Usually we only include the characteristics that are important for design, and rely on naming to convey the full intension. For instance, there is more to being a person than having a name and a birth date, we all know that, but this may be sufficient for the purpose of our model

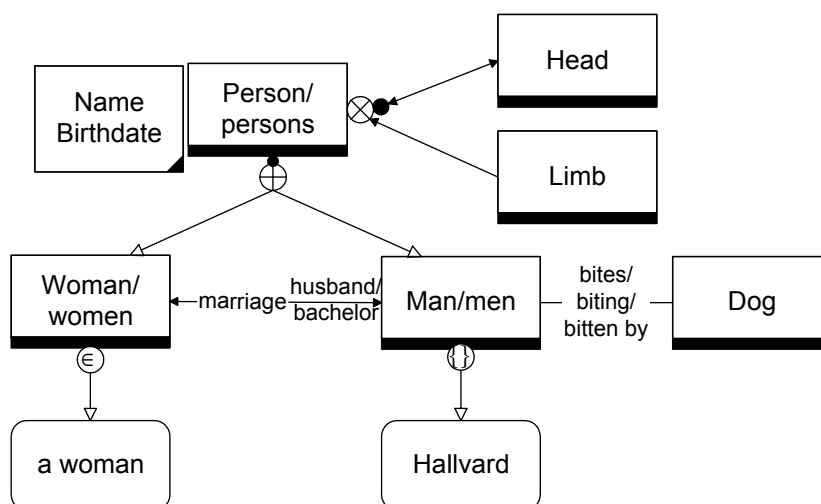


Figure 23. RML example model

The philosophical basis is not unlike the object-oriented view that instances have identity and that a class corresponds to the set of its instances. Hence, a concept is similar to a UML class(ifier) and our instances correspond to UML's objects. However, within object-oriented software design the class is viewed as a cookie-cutter or template from which instances are created. This implies that the relation between an instance and its class is static, and that the set of characteristics is fixed. According to the RML model, however, the classification of a particular instance, i.e. the concepts it is of/belongs to, may change many times during the instance's lifetime. Reclassifying an instance may imply changing the set of characteristics attributed to it, like relations and attributes, but does not change its identity. Such reclassification should be considered a way of updating our model to reflect real-world changes, not forcing a change in the instance. In addition to this distinct philosophical difference between RML and UML, RML supports grouping attributes in several attribute sets, while the attributes in UML all reside in a single flat set.

There are two kinds of instances, specific/particular ones like "Hallvard", and generic ones like "a woman", the difference indicated by the element or curly braces symbol, respectively. *Specific* instances are used to name and possibly classify known instances, and are useful when modelling observations and scenarios. *Generic* instances are used in contexts

where anonymous instances are needed, e.g. in invariant expressions or pre- and post-conditions for tasks where they show up as variables.

The set of instances belonging to a concept's extension depends on how real-world behaviour is modelled in terms of the life-cycle of instances. Take the person concept as example: A real-world person is born, lives and dies. The instance that represents the *idea* of a particular person may however, have a different life-cycle, and in particular may outlive the real person. For instance, the concept or idea of former King Olav still lives, although he does not, so we may choose to think of the RML instance as still existing. In most cases, we think of the RML instance as tracking the real-world life-cycle, but it may also be limited. In the context of task and dialogue models, we usually limit the scope of a concept's extension to the instances accessible for the modelled tasks or dialogues, which in case of the person concept may be the persons included in an organisations payroll system.

Since the extension is a set, an instance is said to be a *member* of the concept's extension. When focusing on the set aspect of concepts, we often refer to instances as "elements". The set view of a concept makes the dynamic nature of instance classification more obvious, since set membership is normally viewed as dynamic and independent of the member elements. Being a member of several sets simultaneously is no problem, and compared to using UML's multiple inheritance feature, gives the modeller greater freedom to attach ad-hoc attributes to instances.

The dual nature of concepts as sets, has motivated a convention for naming, to indicate a focus on either the concept or set nature. The concept is often named using a singular in upper case, e.g. "Person", while the set nature is named in plural using lower case, e.g. "person". When both names are needed, we use the convention of including the extension's name separated from the concept's by a slash ("/"), as we have done for "Person/persons" in the example model.

## 4.6.2 Relations between concepts

Concepts can be related to each other in two ways: through generalisation/specialisation, i.e. is-a relations, and through named relations. Is-a relations introduce more fine-grained concepts and are used to define characteristics that are common to a subset of a concept's extension, in addition to the ones already defined. Named relations are used to relate elements among sets. In the example, we see that 'Man' and 'Woman' both are *specialisations* of 'Person', alternatively that "men" and "women" are *subsets* of "persons". All "men" and "women" have in common the potential of establishing a 'Marriage' relation to "women" and "men", respectively. Note that the example model excludes gay and lesbian marriage, and as we will see, polygamy.

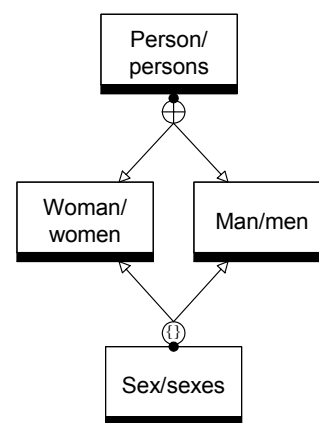
When introducing subsets, we may find it useful to be more specific in two different ways, by expressing that:

1. a set of subsets are (necessarily) disjoint or (potentially) overlapping
2. a set of subsets is a complete partitioning of the (common) superset

In the example, “men” and “women” are defined to be disjoint sets, by means of the ‘+’-sign in the circle, i.e. a person cannot be an element of both “men” and “women” at the same time. We also have expressed that a person is either a ‘Man’ or a ‘Woman’, by means of a black dot at the circle-box attachment point, i.e. that the sets “men” and “women” fully partition “persons”. In cases where there is only one subset in a model fragment, it is a matter of taste whether to use the disjoint (+) or overlapping symbol (<), to indicate the there may be other unmentioned subsets, or just a simple unfilled subset arrow without the circle. As a shorthand, we also allow indicating the superset in the name of a subset, in cases where it is undesirable to include the superset symbol and subset arrow, by appending ‘<superset’ to the name or prepending ‘superset>’, as in ‘Man<Person’ or alternatively ‘Person>Man’. A similar ‘name:concept’ shorthand can be used for indicating concept/set membership, e.g. ‘Hallvard:Person’.

As shown in Figure 30 on page 60, we can make several orthogonal specialisations of the same concept, in this case a ‘Task’. According to this model fragment, a ‘Task’ can be any combination of ‘Action’/‘Supertask’ and ‘Toplevel task’/‘Subtask’, i.e. the cross product of two pairs, since there are two independent disjoint specialisations/subsets.<sup>1</sup> A ‘Method’ is further defined as being both a ‘Supertask’ and a ‘Subtask’, implying that it both contains “subtasks” and is contained in a ‘Supertask’. Orthogonal classification is very useful when modelling several differing points of views, e.g. based on interviews of different user groups. Different classification hierarchies may comfortably reside in the same model, and can later in the development process be integrated if desirable and possible. UML supports this through the use of multiple inheritance and the two disjoint/overlapping OCL constraints. These are however, cumbersome to use for this purpose and we have in fact never seen these constructs used this way.

When introducing a specialisation of a concept, there should be a trait distinguishing it from the general concept, and from the other disjoint subsets, e.g. the participation in a relation. In the example model it is not clear what distinguishes men from women, since the marriage relation is symmetric. It would be necessary to introduce a sex attribute or, as shown in Figure 25, introduce an explicit type concept naming the distinguishing trait. In addition to being subsets of “persons”, “women” and “men” are also specific member of the set of “sexes”. In some cases, we must define the test for classifying elements using a predicate or natural language statement, e.g. a ‘Man’ is a ‘Short man’ if his length is less than 170cm. As a last resort, we can let the social and cultural context provide the distinction. A natural consequence is that although the set of such classes in a model is fixed, the classification of particular individuals need not be. For instance, while “Hallvard” is a ‘Person’ forever and currently a ‘Man’, he may very well become a ‘Woman’ by changing sex, i.e. by moving from the “men” set to the “women” set. The rules and timing for movements among sets is covered by the behavioural model, which will briefly be discussed in Section 4.6.3.



**Figure 24.** Defining sex (types) as distinguishing trait

1. The “full coverage” marker (black dot) on the disjoint subset symbols, in addition express that there are no other alternative combinations.

Relations like ‘marriage’ among “men” and “women” relate pairs of elements from two sets. Each pair is made by forming a binary tuple with one element in each set. All such pairs of a particular named relation constitute a set, which necessarily is a subset of the two participating sets’ cross product. This means that in principle any combination of elements from the two sets can be paired, and an element in one set can be related to zero or more elements in another set, by being (or not) part of several pairs. Relations correspond to associations in UML, but RML do not support “implementing” them as attributes. Relation instances may be included in models, as lines connecting individual concepts, e.g. to state that ‘Hallvard’ is related to ‘Marit’ through ‘Marriage’.

It is possible to express limitations to the general case through three cardinality related constructs:

1. A filled arrow indicates that any element in the set pointed from can be related to *at most one* element in the set pointed to, i.e. the relation is a function and has at most cardinality 1.
2. A filled circle where a relation attaches to a set, indicates that all the elements of that set are related to at least one in the other set, i.e. a full relation membership and at least cardinality 1 in the other end.
3. The cardinality can be explicitly given, for limited cardinalities greater than 1.

In the example in Figure 23, a ‘Person’ must have one, and only one “head”, and zero, one or more “limbs”. Both “heads” and “limbs” can belong to at most one “person”, thus, ruling out Siamese twins. “Men” and “women” need not be married, and if they are it can only be to one ‘Woman’ or ‘Man’ at a time. Individual relations cannot have coverage constraints, since these only have meaning for sets of relations, but may have cardinality to indicate the number of actual relations. In UML’s class diagrams the cardinality is indicated by text labels at each end of the relation, possibly using \* and + to distinguish between the 0 and 1 cardinality case. The reason we have chosen to introduce the arrowhead and circle, is to “force” the modeller to consider these two cases separately: is the relation functional and is it complete with respect to the participating sets.

Relations are sets of tuples and may be subject to specialisation through the subset constructs, and in fact serve as full fledged concepts. The typical usage is restricting the range and/or coverage/cardinality of a relation, for special domains, or providing a special interpretation and name for a specific range. Figure 26 models that in Norway, any pair of individuals can establish a legal dependency relation called ‘Partnership’. However, if the two individuals have opposite sex this relation is called ‘Marriage’, hence ‘Marriage’ is a specialisation of ‘Partnership’, as modelled in the figure. Including circles with ‘+’ or ‘<’ signs and unfilled arrows may clutter the model, so the ‘Marriage < Partnership’ shorthand used for subsets, may as indicated be a good alternative.

Often the relation name will suggest a direction for reading it, although this may lead to ambiguities, like ‘bites’ for “men” and “dogs”. It may be useful to provide up to three names, one neutral and two directional, and we use the convention of separating them with a slash (“/”), as in the example: (man)”bites”(dog)/“biting”(among men and dogs)/(dog is)”bitten by”(man).



In many cases, it is natural to give a specific name to a concept that necessarily takes part in a relation, similar to the role concept used in many modelling languages. For instance, a man that is married is a ‘Husband’ and an unmarried man is a ‘Bachelor’. This can be indicated by adding a name to the attach point, with a slash (“/”) separating the two, as in the example. Figure 26 shows the full meaning of this notation: The ‘Husband’ and ‘Bachelor’ concepts are specialisations of ‘Man’, and ‘Husband’ fully participates in the ‘marriage’ relation. This is similar to annotating associations with role names in UML.

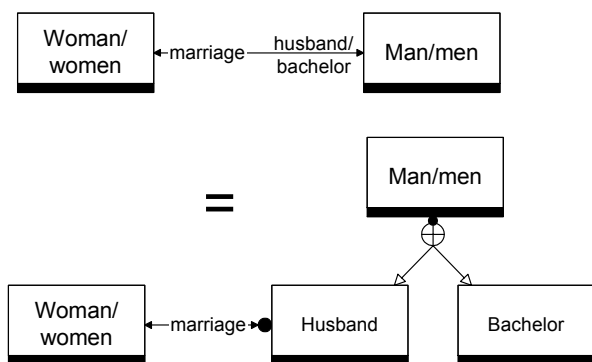


Figure 25. Named relations and membership

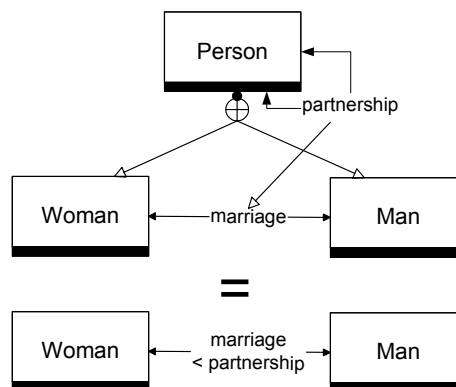


Figure 26. Specialisation of relations

Adding extra names to a model, using either of these naming conventions, may seem like overloading the model. However, it is important to be able to relate everyday language usage to the concepts formalised in a model, to validate the modellers’ understanding of the concepts. Whether a shorthand like “husband/bachelor” or the more verbose model fragment is used, is a matter of taste and practicalities.

*N-ary relations* are modelled using a circle with an ‘x’<sup>1</sup>, and is most often used for the (transitively) irreflexive aggregation/part-of relation. In the example, the (unnamed) relations between ‘Person’ and ‘Head’ and ‘Limb’ are such a part-of relation. Aggregation should be reserved for concepts that are a necessarily integral parts of the containing concept. Candidates for aggregation is typically physical containment and relations you would like to name “has/ownership/belongs-to” or “contains”. Each of the binary relations forming an n-ary relation may have coverage and cardinality constraints. In the example model, any ‘Head’ must belong to a ‘Person’, and every ‘Person’, must have one and only one. When modelling individuals, the aggregation symbol and individual part-of relations may be used to connect them, representing individual n-ary relations. As mentioned, these can indicate cardinality, but not coverage. UML provides a binary aggregation relation, for capturing the special coupling that exists between a container and its parts. Little distinguishes the weakest variant from ordinary relations, while the strongest implies that objects cannot be part of more than one container and in addition should be deleted when this container is. We have similarly included in RML a more restrictive aggregation relation, a diamond in a circle as shown bottom right in Figure 22, which should be used when the relation is acyclic, i.e. the relation graph is a DAG. Our experience is however, that this constraint usually is assumed anyway, and is mainly interesting for supporting reasoning tools.

1. ‘x’ for cross product, since the extension of the n-ary relation is a subset of the cross product of the relation participants.

In some domains, many named relations are defined in terms of others, e.g. an aunt is the sister of a parent. Such *derived* relations can be defined by the sequence of relations that must be traversed from one end of the derived relation to the other. For instance, the ‘Aunt’ relation is the sequence ‘Sister’ and ‘Parent’ or alternatively, the composition of ‘Sister’ and ‘Parent’, often written as ‘Sister o Parent’. Derived relations are useful both for formally defining real world relations and for providing shorthands for commonly traversed sequences of relations. Based on set theory, the coverage/cardinality of derived relations is easily computed. In the latter case, the derived relation is used as a (one-way) *function* from elements to sets, since it is used only in one direction. We are considering using a subset of the XPath language [XPath@2002] for defining functions from elements in a source set to a target set. The idea is to map the RML model to XML’s hierarchical data model by viewing the RML model as a tree with the source set as the root. The stepwise navigation and filtering syntax of XPath may then be used for traversing the relations down the tree. For instance, a function for computing the set of dogs bitten by a woman’s husband may be defined as “function(woman) = \$woman/marriage/biting”. I.e. starting from the woman parameter (indicated by the ‘\$’ prefix), the MARRIAGE and BITING relations are traversed to give a set of DOGS. Since we in the context of task modelling want to focus on the user’s tasks and not the system’s we postpone the discussion of functions to Chapter 5, “Dialogue modelling”.

### 4.6.3 Dynamics

RML focuses on the static aspects of a domain, and currently lacks extensive features for describing the dynamics. We have however, touched upon the subject in our discussion of classification of individuals as instances of concept, or alternatively, elements as members of sets. Such set membership can naturally vary over time, although the degree will depend on the actual concepts in a model and the criteria for classification. Even the relatively static classification in terms of gender is in reality dynamic, as mentioned above. We will interpret a change in set membership as a *state* change, e.g. the sex change operation necessary for turning a man into a woman represents an event triggering a transition from the state of being a man, to the state of being considered a woman. The stippled lines in Figure 27 illustrates the correspondence between moving from one set to another and a state transition, as the result of a reclassification. According to the rules of disjoint subset membership, a person can only be member of one of the subsets at any time, and accordingly, must leave one state and enter the other when the sex change occurs. In the state machine interpretation the transition is labelled with the happening, i.e. the sex change operation. Similarly, a man would transition from bachelor to husband when his wedding occurs, and back in the event of a divorce.

In the Statechart diagram language described in [Harel, 1987] and later incorporated into OMT and subsequently UML, hierarchical and parallel states are supported. Parallel states provides a way of expressing that different parts of a system are mostly independent, i.e. is best modelled by separate state diagrams. The aggregation construct has a corresponding meaning in RML, as illustrated in Figure 28, where the body and mind of a person are modelled as parallel states and separate parts.<sup>1</sup> If combined with the model in Figure 27, we would get three parallel substates, one for the man-woman classification substate and one for each of the two parts, as shown in Figure 29. Each part could in turn be decomposed if desired, and the state diagram would grow accordingly.

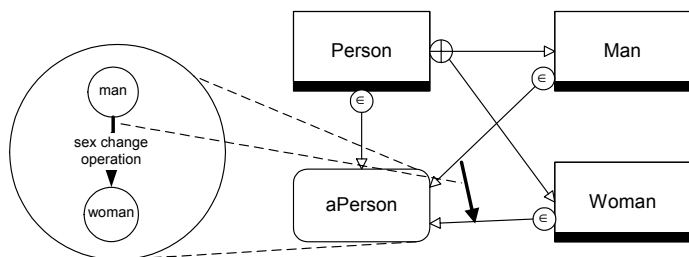


Figure 27. Specialisation and Or-decomposition

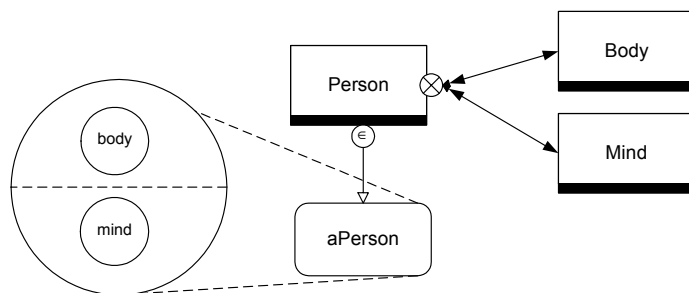


Figure 28. Aggregation and And-decomposition

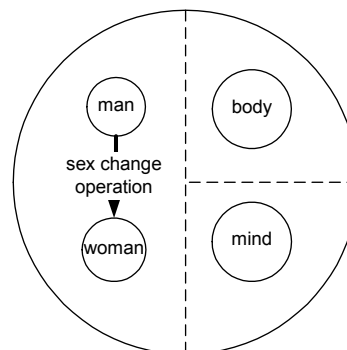


Figure 29. Combining specialisation and aggregation

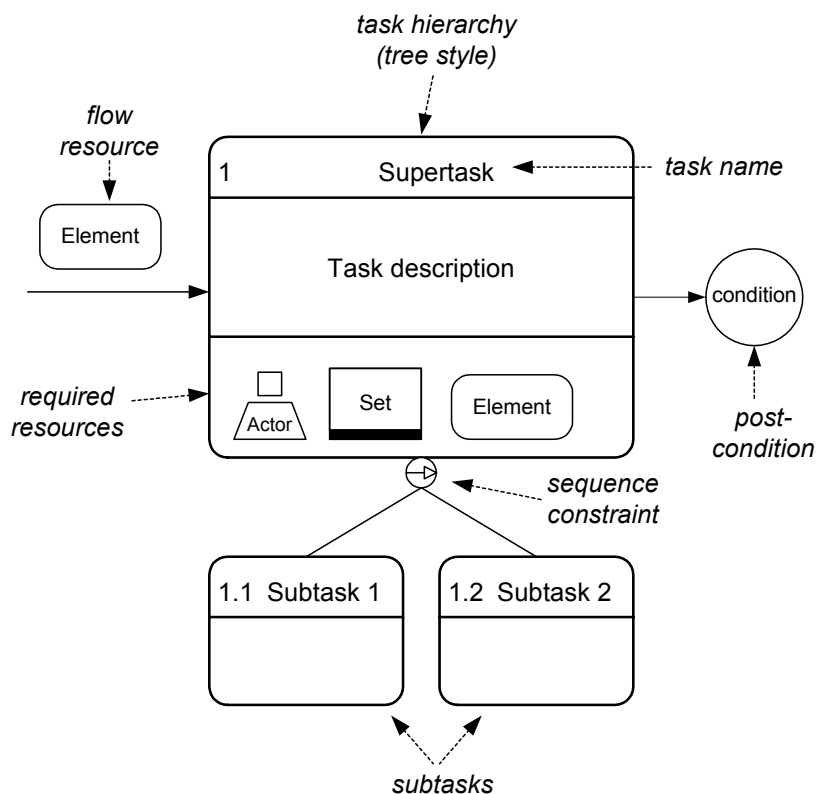
## 4.7 The Task Modelling Language - TaskMODL

The previous sections presented the domain modelling language that is used for the static part of the task domain, and which may be considered the static part of the task modelling language. In this and the following sections, we turn to the dynamic modelling constructs of the task modelling language. Before introducing the underlying meta-model, we take a brief look at the basic language constructs, as shown in Figure 30.

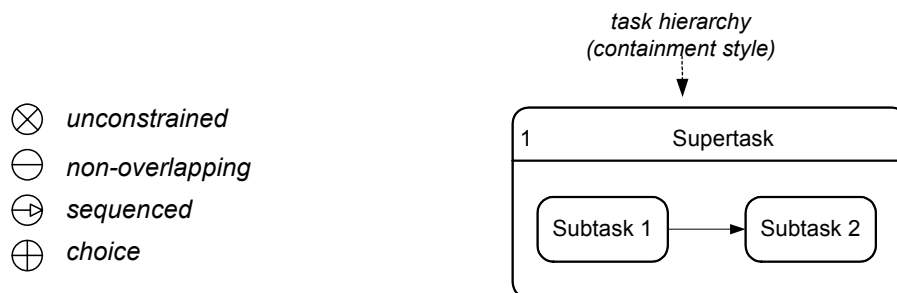
The rounded rectangle is our notation for tasks, and includes a top compartment for the identifier (number) and the name. The middle compartment is optional and may contain a textual description or the task’s decomposition into subtasks. The bottom compartment is also optional and may contain a set of resources that are needed by the task. These resources include the user (type) performing the task and contextual information represented by domain elements and sets. A task may also require information as flow resources, either domain elements or sets of such elements, which are used to indicate triggering behaviour. A task may have an explicit goal or post-condition, which is indicated by the round state symbol, as shown. Alternatively, an output flow may show that certain domain elements or sets will be available as the result of performing the task, but this is not shown in the figure.

A task may be decomposed into a set of subtasks, that may be performed as part of performing the containing supertask. In the tree style notation used in Figure 30, subtasks are

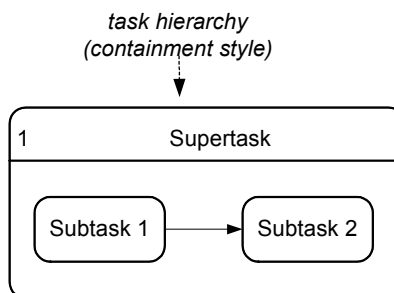
1. The classic Statecharts notation is different, and in particular the parallel substates of an and-decomposition are represented by separate compartment in a rounded rectangle. We have chosen to use circles and introduce an additional level, to make states easier to distinguish from other “boxes”.



**Figure 30.** Basic TaskMODL constructs



**Figure 31.** The four sequence constraints



**Figure 32.** Alternative task hierarchy containment style

included outside their supertask and typically below it. This notation is meant to have the look & feel of traditional HTA-like task models. The subtasks are grouped and connected to the supertask through a small circle attached to the rounded task rectangle. This circle duals as a placeholder for a sequence constraint symbol. The four possible ways of constraining the sequence of subtasks are shown in Figure 31. In the alternative task hierarchy containment style, the subtasks are included in the middle compartment of the rounded task rectangle, as shown in Figure 32. This notation is meant to be similar to dataflow, process and workflow notations. In this case, the grouping is obvious and the small circle is only included for the purpose of constraining the sequence. In Figure 32, a control flow arrow is used to indicate sequence, instead of the constraint symbol for strictly sequenced subtasks.

We now proceed with a deeper description and discussion of the TaskMODL constructs.

### 4.7.1 Tasks and subtasks

*Supertasks* and *subtasks* are specialisations of the general task concept, and the actual classification depends on the task's position within a task hierarchy. Figure 33 shows that these two types of tasks are related through an aggregation relation, and that a supertask necessarily contains one or more subtasks, and that a subtask necessarily is contained in a supertask. In addition, *action* is defined as a non-decomposed task, and for completeness the concept of a *toplevel task* is included. The task hierarchy need not be a strict tree, but it must be acyclic. The former freedom is among others useful in GOMS-like models, where *methods* are expressed as alternative ways of composing a common set of subtasks.

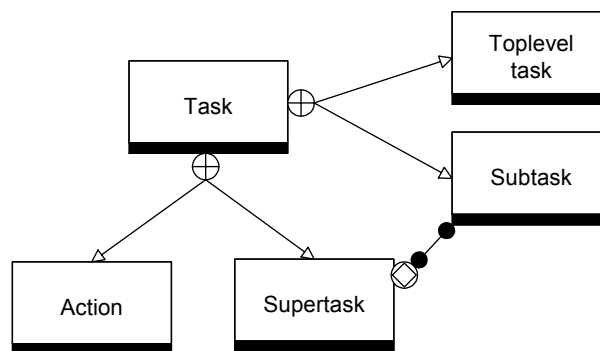
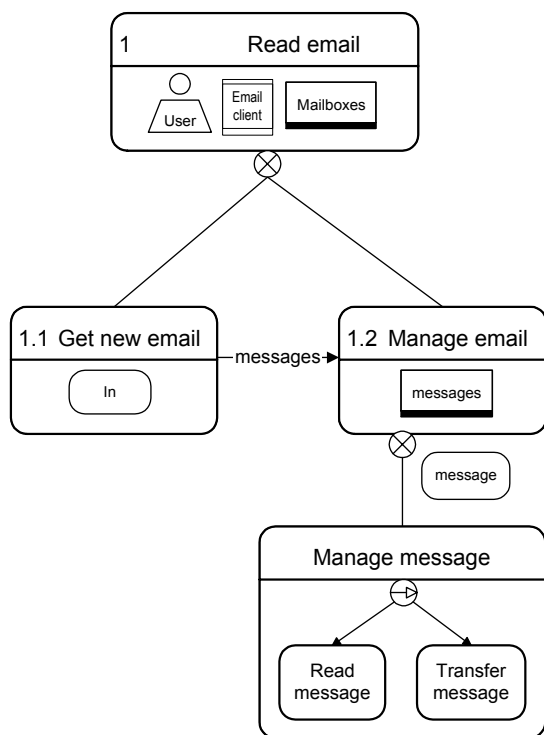


Figure 33. The four basic task types

The task hierarchy need not be a strict tree, but it must be acyclic. The former freedom is among others useful in GOMS-like models, where *methods* are expressed as alternative ways of composing a common set of subtasks.

In the TaskMODL example in Figure 34, we see six tasks, READ EMAIL, GET NEW EMAIL, MANAGE EMAIL, MANAGE MESSAGE, READ MESSAGE and TRANSFER MESSAGE. Both READ EMAIL and MANAGE MESSAGE contain two subtasks. READ EMAIL is classified as a toplevel and supertask, MANAGE EMAIL and MANAGE MESSAGE are both supertasks and subtasks and the other three are subtasks and actions. Two different notations are used for the supertask/subtask relationship, one hierarchical and one utilising containment. In the figure, the first is used in the top-level task, i.e. 1 - 1.1 and 1.2, while the second is used in the bottom task, MANAGE MESSAGE. By supporting both notational styles, TaskMODL should be easier to for both the traditional hierarchical tradition and the workflow-oriented tradition.



- A 'mailbox' contains "messages"
- 'User' performs 'Read email' using 'Email client'
- The current set of "mailboxes" provides the task context
- 'Get new email' uses the "In" mailbox and provides 'Manage email' with a set of messages
- 'Manage email' implies acting on each individual message in the input set
- A message is transferred after it is read

Figure 34. TaskMODL example model

A task model is a static picture of the dynamic relationship between supertask and subtask instances. The dynamic nature of task performance is captured by how the life-cycle of task elements is mapped to the actors' performance of tasks. Each time an actor starts performing a task, a task instance comes into existence. When the task is considered performed by the actor, the task instance ceases to exist, i.e. its life-cycle corresponds to the actor's performance of the task. For subtasks, i.e. tasks other than toplevel tasks, a part-of relation instance is, in addition, established to the task's supertask, with a corresponding lifecycle. At any point in time, there will exist part-of relations linking every actual subtask instance to its supertask.<sup>1</sup> When the subtask ends and a following subtask starts, the first part-of relation vanishes and a second is established. In case of overlapping subtasks, more than one subtask can be part-of related to the common supertask. Section 4.7.3 will introduce constructs for constraining the subtask sequence.

Tasks are performed by actors, when these decide to do so. We distinguish between the possibility of performing a task and the actual act of (starting to) perform it. The former is concerned with satisfying the task's pre-conditions, e.g. that necessary resources are available. The latter is defined by the actor's intention, and corresponds to the creation of the task instance. Similarly, there is a difference between starting to perform a supertask and starting to perform the first subtask of it. Although the subtask often can start at once, when the resource are provided by the supertask's context, the actor decides when to start the subtask, i.e. when the subtask instance and corresponding part-of relation is created.

The relation between RML and TaskMODL becomes even tighter when we consider task classification. Each task instance that exists during the performance of a task, is of course classified as either toplevel/subtask and supertask/action. In addition, the task instance is classified according to the task model, e.g. as a READ EMAIL, GET NEW EMAIL or MANAGE EMAIL task. The effect is that each task in the task model introduces a new RML task concept, as a specialisation of two of the four general task concepts. We see that the meaning of a task model is directly defined in terms of the RML language, in fact, each task model can be seen as a shorthand for a conceptual model in RML. While Figure 33 shows a model of the domain of general tasks, each task model is a model of a more specific set of task, implicitly specialising the general model. Hence, the task model notation can be seen as a task-oriented shorthand for RML, rather than a full language of its own, defined by its own meta-model.

Figure 35 shows how a simple task hierarchy in the task-specific notation is mapped to an RML fragment coupled with the general task model. The 1. SUPERTASK and 1.1 SUBTASK pair on the left is a shorthand for the two specialised task concepts SUPERTASK-1 and SUBTASK-1.1 on the right. The part-of relation within the pair, is correspondingly a specialisation of the general part-of relation between SUPERTASK and SUBTASK, in this case expressing that instances of SUPERTASK-1 can only contain SUBTASK-1.1 instances, or alternatively that SUBTASK-1.1 instances can only be performed in the context of a SUPERTASK-1 instance.

---

1. We see that the (re)use of the aggregation symbol from RML is no coincidence, the symbol connecting READ EMAIL and GET NEW EMAIL really is aggregation.

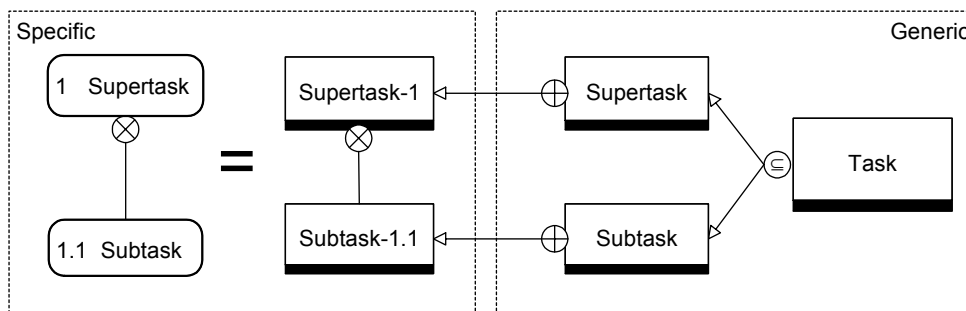
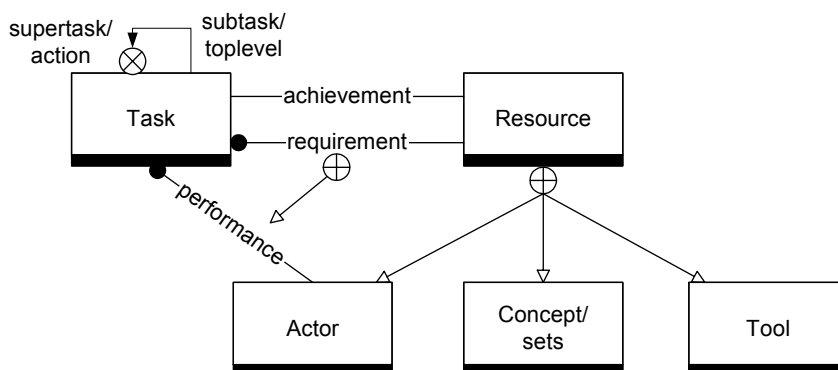


Figure 35. TaskMODL - RML correspondence.

### 4.7.2 Resources

Besides the performing actor, a task can require other *resources*, like presence of information and tools for performing the action. The toplevel READ EMAIL task in Figure 34, is performed by the USER actor using a tool labelled EMAIL CLIENT. In addition, the task is performed in the context of a set of the MAILBOXES, i.e. the current extension of the MAILBOX concept in the domain model of the example. Each task defines the resources it needs, and the resources are pre-conditions for performing the task, hence it cannot be performed until the declared resources are bound to actual ones. In Figure 36, we have augmented the generic task model with three different resource types, actors, tools and the information context. Note the more compact form of definition for the four task types.

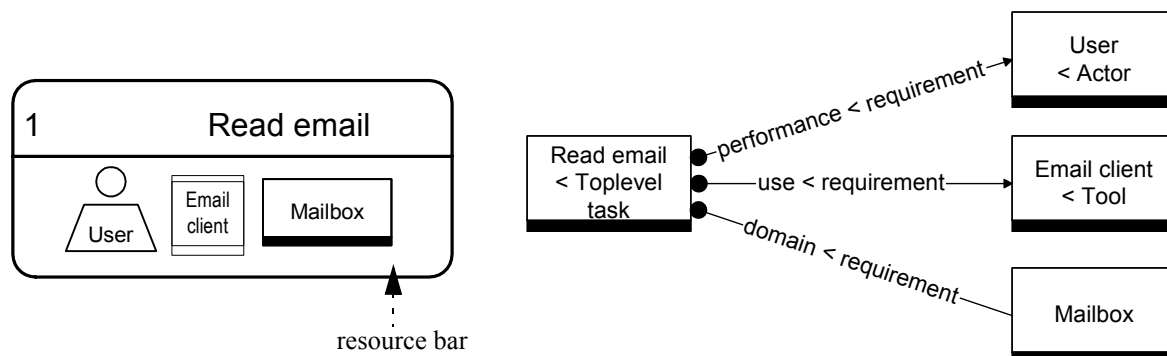


- Tasks are hierarchically structured.
- Tasks have pre- and post-conditions, i.e. requirements and achievements.
- Conditions are specified in terms of resources, which are required for performing tasks and made available by performing them.
- Actors are required for performing of tasks.
- Information and tools are additional resources that may be required for performing tasks

Figure 36. Generic TaskMODL model

The *requirement* relation between task and resource acts as pre-conditions and can, for each kind of resource, i.e. actor, information and tool, be interpreted as ‘performance’ (as indicated), ‘context’ and ‘uses’, respectively. When including resources in the bottom resource bar of a task, as for READ EMAIL, this relation is implicitly specialised, to indicate that this particular kind of task requires these specific resources. Figure 37 shows the corresponding

RML model for the READ EMAIL task. The USER and EMAIL CLIENT resources are specialisations of the appropriate resource concepts, and have specialised requirements relations. The small filled circles at the left attachment point indicates that the relation is indeed required, as it specifies that the cardinality is at least one.



**Figure 37.** Interpreting the resource bar notation as an RML model fragment

The *achievement* relations represent the goals or results of performing a task, i.e. post-conditions, and consists of a set of resources in a particular state, which are made available after performing a task. These resource relations are established during the task's performance and hence, are available when the task is considered finished.

*Actors* are the entities that perform tasks, and are defined in actor models that are structured in classification and instance hierarchies. The actor model has two main uses. First, the hierarchy of concrete actors models the actual user population, and the organisational structures they work within. This part of the model is typically used when representing observed work practice, to indicate who does what and how. Second, the abstract actors or roles are defined in the classification hierarchy, where the key user types are identified and their characteristics defined. I.e. the classification hierarchy provides intensional definitions of sets of actors corresponding to user groups/classes/stereotypes. The specialisation mechanism is the same as used in the conceptual modelling language, i.e. based on groups of subsets, which can be disjoint or overlapping. The two hierarchies will be linked by the classification of the concrete actors. For design, it is important that the model of the abstract actors focus on design-oriented characteristics and abilities, like experience and cognitive skills, and not more abstract or less relevant characteristics like age, size and gender. The model of the concrete actors gives an impression of (the relative) size of each set of abstract actors and is useful for setting design priorities. Figure 38 shows a typical actor model containing both hierarchies, with the concrete actors on the left and abstract actors on the right.

The meaning of the actor model is defined by a transformation to a RML model. As for tasks, actors are implicitly interpreted as concepts (and instances) extending the generic model shown in Figure 40 in Table 6. As in APM, actors with round heads are abstract and correspond to RML concepts, while those with rectangular heads are concrete and correspond to RML instances. The RML fragment corresponding to the example in Figure 38 is shown in Figure 39. Table 6 summarizes the resulting four actor types and the corresponding RML model fragment. Note that since actors are resources, they can be used as task parameters and information flow, e.g. as the result of an allocation task. In this case a concrete actor symbol will be used, since it refers to sets or elements which are actual/existing users, and not roles.



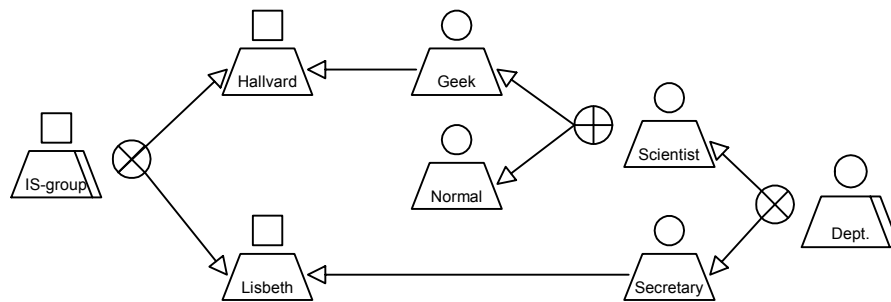


Figure 38. Typical usage of actor modelling

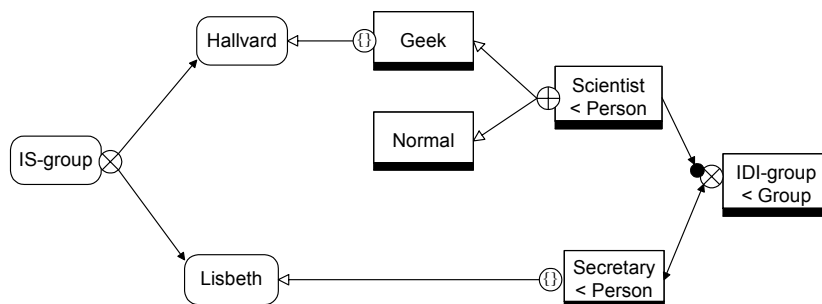


Figure 39. Corresponding RML model fragment

Actor types	Concrete	Abstract	RML model
Simple	A real, named <i>person</i> will perform this task. Corresponds to a particular instance of 'Person'.	A person playing this <i>role</i> or having these <i>characteristics</i> will perform the task. Corresponds to concept specialising 'Role'.	<p>Figure 40. Generic actor model</p>
Composite	One or more persons from this named <i>group</i> will perform this task. Corresponds to concepts/sets specialising 'Group'.	Persons playing these <i>roles</i> will perform the task. Corresponds to concepts/sets specialising 'Person'.	

Table 6. The four actor types

*Tools* are the elements that the user interface is constructed from and correspond to abstract interaction objects or dialogue elements or structures of such. Tools are structured in an aggregation hierarchy, normally mirroring the task hierarchy. When a task model describes current practice, the tool resources link to the dialogue elements used for performing the tasks. When designing new tools, the tool resources record design decisions, i.e. provides a link from requirements to design. We will return to the tool concept when discussing dialogue modelling in Chapter 5.

*Domain elements*, i.e. concepts/sets, instances/elements, relations and attributes, are used both as contextual information and operated upon. Contextual information is typically used as mental support, e.g. for making decisions about which tasks to perform. When operated on, domain instances are typically transformed, run through a series of states and otherwise modified, including created or destroyed. As resources they will eventually be bound to actual sets, instances and data. Hence, in this context the extension of a concept implicitly

means the actual extension at the time of performing the task, i.e. those elements that are available for relating to the task. The integration of the domain and task model is straightforward, since the task model can be transformed to an ordinary RML domain model, as described above.

In the context of task models, RML model fragments may refer to a more limited universe of discourse than when used alone. For instance, the ‘Person’ concept may in an RML model refer to every dead, living or future person. In a task model used for describing actions within an organisation, the same concept may instead refer to the all the employees and customers of the organisation. Hence, the ‘universe’ is implicitly reduced to the organisation and its surroundings. There will also be a need for referring to arbitrary subsets of a concept’s extension, e.g. a set of instances containing only “persons”, as distinguished from the all “persons” in our universe. In the context of task models, we may introduce conventions of like using the extension name, e.g. “persons”, for referring to arbitrary subsets and the concept name, e.g. “Person”, for referring to the full extension. When we need to refer to singleton sets, we may similarly use the instance/element symbol and the singular name in lowercase, e.g. “person”. Such conventions may reduce the need for specialised constructs in the language, and make it easier to use for those who know the conventions. On the other hand, tools that are not programmed to use these conventions will not be able to correctly interpret models using them.

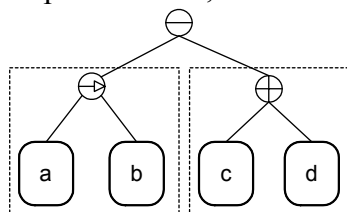
### 4.7.3 Subtask sequence constraints

The set of subtasks of the supertask may be sequentially constrained by introducing data or control flow or explicit sequence relations. In the example model in Figure 34, we see that GET NEW EMAIL provides input to MANAGE EMAIL, and hence must precede it. The two tasks READ MESSAGE and TRANSFER MESSAGE are sequenced, too, by means of an explicit sequence relation. As shown in Figure 31, there are four different sequence constraints (the symbol in parenthesis indicates notation):

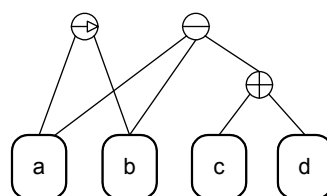
- Aggregation (x cross): No constraint at all, meaning any order of overlapping subtasks. The aggregation symbol is used for the tree-like notation, to connect the supertask to its subtasks.
- Order (- dash): Indicates non-overlapping subtasks, without any specific order constraint. More precisely, it means that the supertask cannot participate in more than one part-of relation with a subtask at a time, i.e. the part-of relation is in practice binary. This sequence relation is mostly used to stress the non-parallel nature of the subtasks.
- Sequence (> arrow): An explicit sequence is given, usually based on graphical left-right or top-bottom ordering. In terms of the part-of relation, this not only means that it is binary, i.e. implies order, but also expresses that the subtask end of the part-of relation will go through the same specific sequence for each performance of the supertask. In other words, the supertask goes through a sequence of states, each requiring a part-of relation to a specific kind of subtask. The actual reason for the specific sequence may be data dependencies, mental or physical constraints, or just an observed habit. In case of data dependencies, a special dataflow notation may be used, as shown in Figure 34, where a set of messages “flow” from left to right.

- Choice (+ plus): Either-or constraint, meaning that only one of the related subtasks will be performed during the current performance of the supertask. I.e. the supertask will only participate in one part-of relation throughout its lifetime. This relation has an important use for classification, as will be discussed later. Choice could be modelled as two (or more) optional subtasks with complementing (disjoint) conditions, as discussed in Section 4.7.4, but a special construct is more expressive. To better support the design process, each arrow can be annotated with a percentage, to indicate how relatively often each subtask usually will be performed.

The four sequence constraints are of *strictly increasing* strength, so replacing e.g. order with sequence will limit the number of different task sequences. It is important to note the order and sequence constraints only provide constraints in one time direction. I.e. they state that some subtask cannot be performed until a certain time, but say nothing about how soon or late they are performed, if performed at all. In particular, other tasks from other parts of the task hierarchy may be performed in between, and as we will see when discussing task classification and specialisation, new ones can be inserted.



**Figure 41.** Example 1:  
order(sequence(a,b),  
choice(c,d))



**Figure 42.** Example 2:  
sequence(a,b) and order(a,b,  
choice(c,d))

To express a specific complex constraint, the relations may have to be combined in a directed acyclic graph, usually a strict tree. In Figure 41, subtasks a, b, c and d can be performed in one of the following sequences: a-b-c, a-b-d, c-a-b and d-a-b. If c or d in addition was to be allowed to occur in between a and b, we would have to make a constraint graph, as shown in Figure 42.

A constraint tree of the former kind, is equivalent to grouping each subtree into subtasks, containing one sequence constraint each, as indicated by the stippled box in Figure 41. The possibility of introducing such anonymous/invisible subtasks, greatly simplifies modelling, since one avoids cluttering the model with artificial subtasks and inventing names for them. Since an arbitrary name may be worse than no name, explicit anonymous/untitled subtasks are allowed, too.

The four sequence constraints are illustrated by the model shown in Figure 43. The example uses anonymous tasks to simplify the model. The USE PHONE task model expresses that:<sup>1</sup>

- The toplevel USE PHONE task is a sequence of two anonymous subtasks.
- The first subtask is a choice between a CALL or RECEIVE task.
- The second subtask consists of an unconstrained and possibly overlapping combination of STORE NR. and CONVERSATION tasks.

1. This particular model resulted from an elicitation session in a lecture on task modelling.

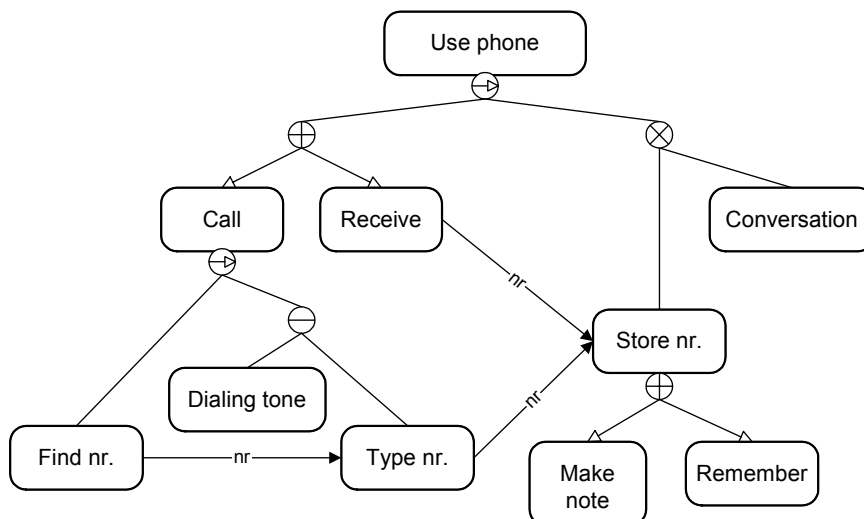


Figure 43. Use Phone task example

- The FIND NR. task must precede the DIALING TONE and TYPE NR. tasks, and the two latter may not overlap. In addition, the number that TYPE NR. requires, may result from performing the FIND NR. task.
- The STORE NR. task requires a number, which may be provided by either the RECEIVE or the TYPE NR. tasks. Since these two latter tasks are in disjoint subtrees, only one of them will be performed, and hence provide the value.
- Storing a number is performed by either a MAKE NOTE or a REMEMBER task.

#### 4.7.4 Subtask cardinality

The part-of relation is n-ary and can include several members of each related set. For instance, in the RML example model, a person can have several limbs. In the general case of the supertask subtask part-of relation there is no specific cardinality constraint, unless specifically introduced. In many cases it is relevant to limit the subtask cardinality. For instance, children are taught to look in both directions along a street until it is clear two times in a row, before crossing it. In some cases, the cardinality may depend on contextual information, e.g. a task may be performed once for each element in a set that is provided as a resource, to give the effect of iterating through the set.

Cardinality constraints are expressed as for ordinary relations in RML, i.e. full coverage is indicated using the small filled circles, and limiting the cardinality to one is indicated using the filled arrows. A specific cardinality may also be directly specified using a number or interval, and conditions, rules or formulae for determining actual cardinality may be added. In the case where no sequence constraints are specified, the cardinality constraints have the same meaning as in RML. However, when specific sequence relations are used, the cardinality constraint will be given a different interpretation. All the sequence constraints besides aggregation (x) already limit the cardinality to at most one at a time. Hence, there is little point in adding cardinality constraints. It makes more sense to relate such additional cardinality constraints to the *lifetime* of the supertask, rather than to any *instant* in time. Our interpretation of a cardinality of n is similar to having n explicit part-of relation in the model. However, it must be taken into account that all the subtasks connected to a particular

sequence constraint may have cardinality constraints. This gives the following interpretations:

- Aggregation: The standard constraint, i.e. at no point in time will more (or less) than this number of the subtask be performed.
- Order: The cardinality constraint is related to the lifetime of the supertask. I.e. during the performance of the supertask, no more (or less) than this number of the subtask will be performed. Note that the other subtasks connected to the same order symbol may be interleaved with these ones.
- Sequence: Similar reasoning to the order relation, i.e. no more (or less) than this number of the subtask will be performed together in sequence, and in this case, not interleaved by other subtasks.
- Choice: If chosen, the subtask will be performed more (or less) than this number of times.

The cardinality constraints may very well be computed based on contextual information, like the size of sets or value of instance attributes. A variant of the former is shown in Figure 34, where the number of MANAGE MESSAGE subtasks is constrained by the number of message instances available. I.e. each subtask instance will have a corresponding message instance as an information resource. The general way of doing this is shown in Figure 44. In this model fragment, the SET TASK takes a set as input, and is decomposed into a set of tasks taking an element as input, giving one element-oriented subtask for each element in the set. The naming of the set and the element is the same, to implicitly express what is also explicitly expressing using the element-of notation (encircled e).

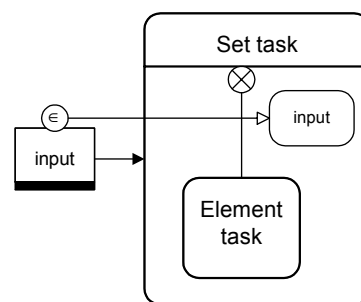
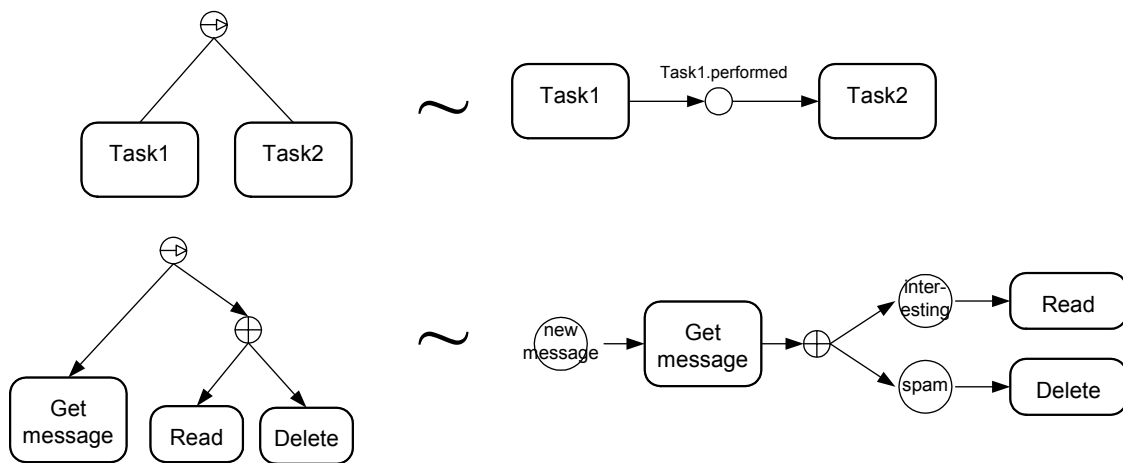


Figure 44. Decomposing set- and element-oriented tasks

Repetition or iteration of a task can be achieved with cardinalities greater than 1, which is the reason we have no explicit looping construct in TaskMODL. A special case of constrained cardinality is optional tasks, where a condition determines whether a task is performed or not. This condition in effect distinguishes between the 0 and 1 cardinalities. A subtle point is worth noting: an optional task may very well be initiated, being optional means that it will not necessarily be completed or alternatively, that its post-condition will not necessarily be satisfied. This interpretation is similar to allowing a task to be interrupted, which ConcurTaskTrees handles with a special interrupt operator.

### 4.7.5 Explicit pre-conditions and post-conditions

The sequence constraints provide a limited way of specifying under what circumstances a task may be performed, i.e. conditions that must be satisfied before performing a task. These conditions all refer to task performance and timing, e.g. that a certain task has already been performed, and each constraint may be expressed as a (composite) precondition in terms of the performance of some tasks. There is also a need for expressing more general conditions in terms of the domain, by referring to the extensions of the concepts and individuals defined in the domain model.



**Figure 45.** Sequence constraints and corresponding control structure using conditions

First, there is a need for making explicit the post-conditions by formulating the result of performing a task, besides the fact that it has been performed. Second, we need to model the motivation for a particular task sequence, by using the post-conditions of one task as pre-conditions for a following task. When a task is triggered by another, the deeper reason may be the state of the domain achieved by the first task. If this state is entered for some different reason than performing the first task, the second task may be performed even though the first one is not actually completed or performed at all. Figure 45 shows examples of sequence constraints reformulated using explicit conditions or states. In the top model, a binary sequence constraint is translated to a post-/pre-condition in between the two tasks. In the bottom model, sequence constraints are made more explicit by introducing labelled conditions. The next step would be to formalize the conditions, e.g. define what “new message” really means in terms of the domain model.<sup>1</sup>

By using the domain model, new kinds of conditions are possible to express, e.g. a task may be relevant only when a particular individual or relation comes into existence, or when an attribute of any or a certain individual attains a particular value. In general, there will be a need for both referring to the current state of the domain and comparing the current with a past state for noting changes, i.e. creation, reclassification or deletion of individuals and relations, and change of attributes. We have chosen to use RML model fragments to formulate states and chains of states to express change events. The arbitrary element symbol is used for defining variables and existential qualification. In many cases, the pre-condition fragment will include a qualified variable that is used as input to a task, and in this case there is no need for an explicit state symbol. In other cases, the domain model fragment will be enclosed in a condition symbol (circle) which will be linked to an enabled task or following state.

Figure 46 shows two examples of pre-conditions. The top fragment shows that the MARRY task requires (possibly as an actor resource) a MAN individual, which must also be a bachelor, while in the bottom fragment the MARRY AGAIN task requires MAN to first have been a husband, then become a bachelor, before the task can be performed. Note that with differing

1. RML currently has no constraint language like UML's OCL, but we are considering using XPath [XPath@2002] as a starting point, as it is set-based and is supported by many open-source tools.

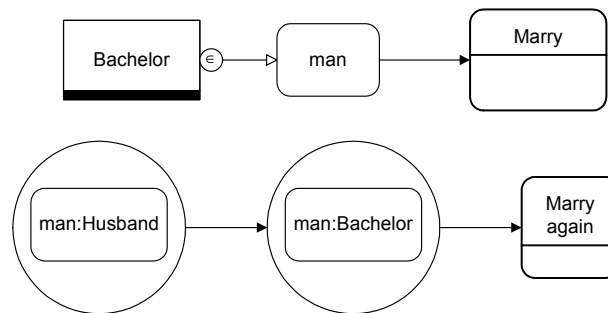


Figure 46. Pre-condition examples

variable names in the two states, the meaning would be totally different, i.e. only requiring that some HUSBAND and some BACHELOR exists, without being the same individual.

### 4.7.6 The task life-cycle, resource binding and data flow

All tasks besides the toplevel task are performed in the context of a supertask. The subtask may only be performance during the performance of the supertask, with the actual timing depending on the explicit sequence constraints present in the model, and on the availability of the required resources. Note that a task may be a subtask within several different supertasks, and hence may be performed in different contexts. Each context must provide the resources required by the task, in addition to satisfying the local sequence constraints.

When an actor starts performing a subtask, the required resources must be *bound* to the subtask. This is a natural consequence of how a task model is interpreted as a referent model fragment, and in particular how the resources are interpreted as relations between the subtask and appropriate instances. I.e. resource *binding* corresponds to establishing the *pre-condition* relations shown in Figure 37, and happens when the task instance is created, based on information in the task's context. In some cases, the binding is static, i.e. the set of elements for which a pre-condition relation is established is pre-determined in the model. For instance, within an organisation some tasks will always be supported by the same tool, so the tool will be a constant in the model. While there technically is no difference between statically and dynamically bound resources, since they anyhow are created just-in-time for the new task, the static ones are normally placed together in the resource bar of the task. In the dynamic case, the resource will be derived (dynamically) from the supertask context and is typically indicated using dataflow (arrows).

The resources that are bound to a subtask, can be provided by its context, in two ways:

1. The current resources of all the supertasks within which the new subtask will be performed, can be used by this subtask. Think of the subtask as being performed inside a set of semi-transparent supertask containers. To locate its resources, the performing actor looks outwards at and through each supertask, until it finds a suitable candidate for each resource. The nearest resources are more visible and hence more likely candidates, while the distant ones are less visible, may be shadowed and are considered less relevant. If the resource bar of each supertask is thought of as its parameter list, this is similar to how static or lexical scoping is used in programming languages, where inner constructs see the bindings in outer ones. When a subtask uses the same resources as those defined in its supertasks, we usually do

not bother to declare them explicitly. However, for supporting reuse of tasks in a new context, it is important to declare the requirements explicitly, so it can be ensured that a set of supertasks provides a suitable context for the subtask.

2. The preceding subtasks provide a more dynamic context for the new subtask, through the results of performing them, as defined by their post-conditions. These conditions typically express that some element or configuration of elements exist in a particular state, and represents a guarantee about the domain for following subtasks. In the model, this can be explicitly expressed by inserting the circular state symbol in between subtasks, or using the dataflow arrow, as used for the “messages” resource from the ‘Get new email’ task, in the example. The supertask can be thought of establishing a set of temporary bindings with local scope, which are set to the result of corresponding subtasks, when these are ready.<sup>1</sup>

Note that there is no notion of “data transfer” from one task to another, ruling out the traditional interpretation of dataflow. The binding of resources is indirect through the task’s context, i.e. its supertasks. The motivation for this is twofold: First, although one task may be triggered by another, nothing prevents the actor from waiting before performing it, if at all. Hence, opening a “data channel” for the transfer of information from the first task to the second, makes little sense. Second, we do not want to rule out other means of providing resources, e.g. because of unexpected or exceptional events. By always looking in the context for resources, the actor (or modeller) is given more freedom for alternative ways of making them available. This freedom may also be used in specialised versions of a task, as discussed below in Section 4.7.7.

Splitting dataflow into a sequence part and an information availability part is similar to how [Malone, 1999] models dataflow. In the context of business processes, *flow dependencies* are “viewed as a combination of three other kinds of dependencies: *prerequisite* constraints (an item must be produced before it can be used), *accessibility* constraints (an item that is produced must be made available for use), and *usability* constraints, (an item that is produced should be “usable” by the activity that uses it). Loosely speaking, managing these three dependencies amounts to having the *right thing* (usability), in the right place (accessibility), at the right time (prerequisite).” In our task-oriented context, the reasoning is similar, resources are produced by tasks, made available to other tasks through their common context, and validated by the actor before being bound to future tasks, although the last step is not made explicit in our approach.

In many cases, the actual resources used by a subtask differs from its supertask’s, and must either be derived from those provided, e.g. by a function, or be declared explicitly. In the example, the GET NEW EMAIL task provides its own context, which is the IN mailbox instance from the domain model. The MANAGE MESSAGE TASK derives its MESSAGE resource from the MESSAGES set of its supertask. There are two ways of deriving resources, corresponding to the two kinds of relations we have.

---

1. The programming language model mimics the two main binding constructs in Scheme. “let” establishes a set of bindings with values, so contained constructs can use these at once. This is similar to how the resource in the supertask’s resource bar are immediately available for the subtasks. “letrec” on the other hand, established the bindings but not the values, hence, they must be given values before being used. This is the way the results of subtasks are handled, since the scope is introduced by the supertask, but the values are provided later by the contained subtasks.



1. When specialisation is used, a subtask is declared to use resources of a more specialised type than the supertask, as a filter on the available resources.
2. When named or n-ary relations are followed, the resources used by the subtask are those related to the supertask's resources through the particular relation. Since derived relations can be used, quite complex functions can be used for deriving resources.

The *typical* ways of deriving resources will depend on the kind of resource:

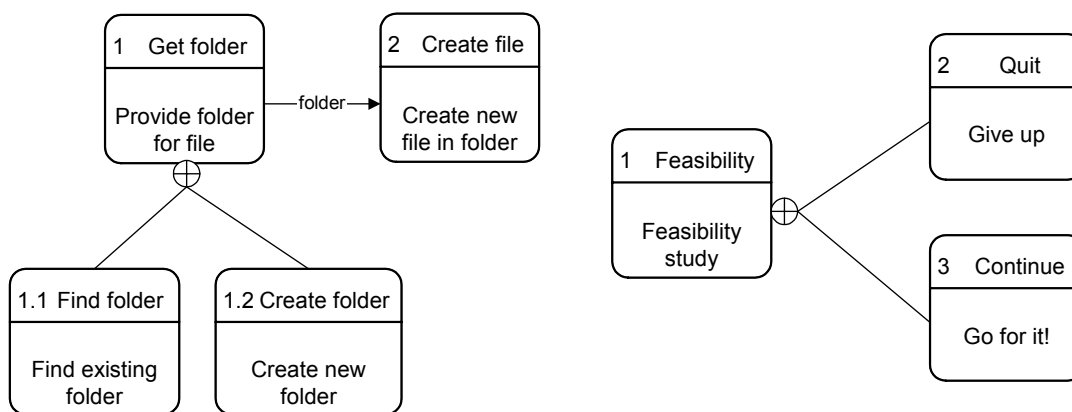
- Actor resources are typically derived by traversing *down* the actor hierarchies, i.e. specialising or detailing the actor resource. In a workflow or high-level task model, it will also be natural to traverse relations among actors, e.g. power relations can be used for formulating delegation.
- Deriving one tool resource from another is typically used to select a concrete tool from an abstract specification of the tool's function. The choice can be made by the designer, the tool itself or the user, depending on the design, and it may be more or less formalised in a rule that the user/designer uses for selecting the tool for performing/supporting the task. For instance, the user may choose to send a short message using the ICQ instant messaging client instead of the mobile phone, if the receiver is expected to be continuously connected to the internet.
- The possibilities for deriving domain elements from a context are partly given by the conceptual language constructs and partly by the actual domain model. From a given concept or set, one can derive subsets of particular specialisations and the members of the extension/set. Using relations, including derived ones, one can derive subsets based on participation, e.g. deriving the husbands subset of a set of men, and sets based on traversal, e.g. deriving the wives of a set of men/husbands. If these mechanisms are insufficient, functions for complex traversal and filtering can be defined.

As discussed above, resources are used for expressing pre-conditions for performing tasks. Some resources, like actors, usually do not change within the same context, since most subtasks in a task model are performed by the same actor as the top-level task. Other resources, especially information resources, will necessarily vary from each performance of a (sub)task. The crucial point in both cases is that the resource must be present at the time the actor wants to perform the task, when it is bound to the task instance. Being present, in turn, means that the resource is available in the task's dynamic context, i.e. its supertask or further up the task hierarchy.

### 4.7.7 Task classification

The choice (+) sequence relation is used to indicate that several subtasks are mutually exclusive, i.e. either can be performed, but only one of them. The reason why only one of the subtasks can or will be chosen, depends on the actual situation. There exists several different usages of this construct:

- The selection of the actual task to perform is deterministic, and is given by an explicit condition. The actor is seemingly given no choice, but it may be that the condition is just a formalisation of the actor's own task knowledge. The point is leaving the choice to the supporting tools and not the performing actor, Or rather, making them capable of choosing on behalf of the actor, a kind of actor - tool delegation.
- Each task represents one way of reaching the same goal, i.e. the choice relation represents method selection. The choice will depend on the nature of the available resources, constraints on the time left for the task, the quality needed or cost of performing each task, or just a matter of the actor's preference or mood. Method selection is used when several effectively equivalent methods have different efficiency or ergonomic characteristics. Although the choice may possibly be encoded and utilized by supporting tools, it should always be up to the performing actor to decide.
- The subtasks have different goals, and the choice is a matter of making a strategic decision. The distinction between this and the previous case is the well-known question of “doing the thing right vs. doing the right thing”. The performing actor will make a choice, depending on the context or other conditions, and since the goals will differ, the following tasks will generally do so too, and it may be most practical to flatten the hierarchy, as shown in Figure 47.
- The supertask represents an abstract and possibly incomplete tasks, and the subtasks are specialisations of the generic supertask. In this case the commonalities of the subtasks are explicitly represented as part of the supertask, and only the differences are defined in the subtasks.



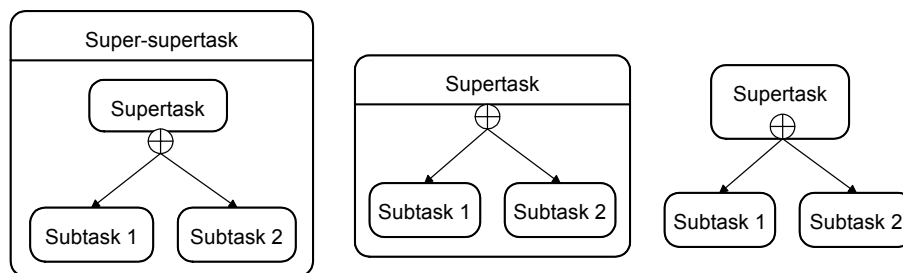
**Figure 47.** Task classification

- *Left:* Same goal, but different methods
- *Right:* Different goals

For all these cases, there must exist a decision procedure for selecting the appropriate subtask, although the procedure may not be possible to express or formalise. The choice symbol suggests an analogue to the disjoint subset construct of RML, and indeed, the choice structure will often mirror a similar structure in the domain or actor model. In a task oriented development method it is natural that distinctions that are important for task performance, are made explicit in the domain model. For instance, in many contexts and cases, there are different ways of handling men and women, so these two disjoint subsets of per-

sons, can be motivated by the task model, as well as conceptually. Similarly, if two sets of concrete actors have consistently different ways of acting, it is natural to define corresponding roles in the actor model. In the task model, these roles will be used in the resource bar in mutually exclusive branches in the task tree.

An important aspect of the choice relation is the context the choice is made in. During task performance, the choice will be made within a task context, and the chosen task can be performed as soon as the necessary resources are available. In fact, sometimes the available resources will decide which choice to make. However, to reason about the availability of resources, the choice must be made in the context of the supertask, not the subtask. This distinction is made by including the choice relation within the supertask, instead of abstracting it away. In figure 44, three model fragments, each with a choice relation, are shown. The rightmost notation, with the choice symbol *inside* the supertask just above in the hierarchy, is used to narrow the scope for making the choice. This is equivalent to the middle fragments, where the choice is made in the context of the immediate supertask of the two mutually exclusive subtasks. However, in the leftmost fragment, the choice is made one level above, in the super-supertask.



**Figure 48.** Choice context

- *Left:* The choice is made in the context of the super-supertask.
- *Middle and right:* The choice is made in the context of the immediate supertask.

Narrowing the scope is desirable for several reasons. First, it makes the model easier to understand. Second, it promotes model reuse by reducing the interaction between fragments. Hence, the middle or right notation is preferable.

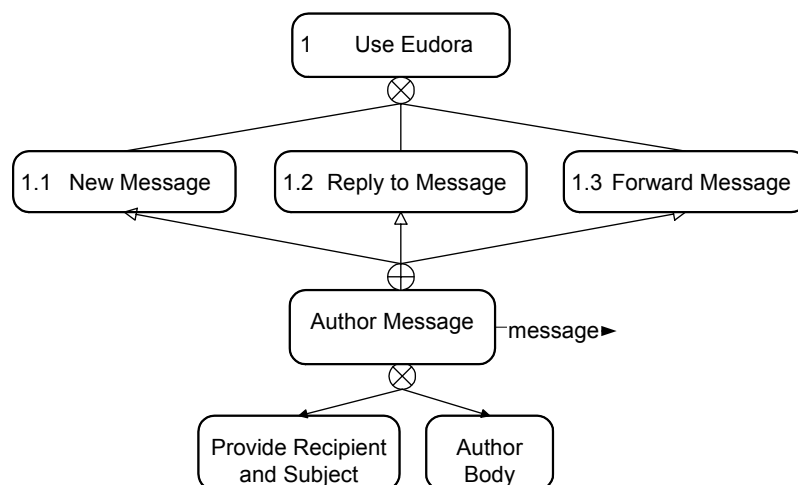
### 4.7.8 Generalisation and specialisation

The notation used in the middle fragments of Figure 48, indicates that two mutually exclusive subtasks can be performed when performing the supertask. The border around them indicates two important facts:

1. The choice must be made in the context of the immediate supertask, and because of the rules for inheriting resources, in the context of resources common to both subtasks. This means the choice is abstracted away from the outside of the supertask, i.e. that the choice of subtask is due to local conditions, e.g. some characteristics of available resources.
2. The subtasks must be equal for the purpose of the supertask's context, i.e. they have the same pre- and post-conditions. This means that the supertask can be seen as a generalisation of the two subtasks, or alternatively, that the subtasks are specialisations of the supertask.

If the subtasks in addition to these two points, have a common internal structure it is natural to consider them specialisation of a single general case, i.e. an instance of generalisation/specialisation structure. The commonalities will be part of the definition of the general task, while the differences will be described in each subtask.

In Figure 49 a quite common case is shown, where a single application, in this case the Eudora email client, provides several related functions, which from a task perspective gives rise to similar task structures. Since the NEW MESSAGE, REPLY TO MESSAGE and FORWARD MESSAGE tasks are so similar, the AUTHOR MESSAGE generalisation has been defined. As can be seen, this task provides a message element and consists of two subtasks, which by definition must be performed as part of the three specialisations. However, each of these may perform the two subtasks differently. For instance, the recipient of a reply is normally the sender of the replied-to message, while a forwarded one introduces a new recipient. In addition, both these will usually have a subject and body which includes part of the subject and body of the source message, respectively. The NEW MESSAGE task, on the other hand, will provide a fresh subject, an arbitrary recipient and a body authored from scratch.



**Figure 49.** Generalising several common subtasks of using Eudora

Generalisation and specialisation has both a formal and a conceptual meaning. As before, the formal meaning of the construct is partly defined by the transformation to an RML fragment connected to the generic task domain model shown in Figure 35. In this case the transformation is very simple, choice corresponds to disjoint subset, i.e. the RML concepts introduced by the supertask and subtasks are related through the standard RML specialisation construct. In addition, all the task sequence constraints must be valid in each specialising subtask.<sup>1</sup> For the example in Figure 49, this only means that each specialisation, must include the same two subtasks. In general it means that the actual ordering of the specialisation's subtasks must satisfy the constraints defined in the generalisation, in addition to new ones that are introduced in each respective specialisation.

For a specialisation to make sense conceptually, there must be some deeper commonality that makes it natural to view a set of subtasks as the specialisation of the same general

1. Although this cannot be expressed in RML, it is a natural consequence of how invariants for a general concept must hold for the specialised ones.

supertask. First, the commonality must have some significance, since any set of tasks can be generalised, by introducing a common supertask with no structure! Second, although it is possible to use anonymous generalisations, indeed any use of the choice relation can be interpreted as generalisation, it should be easy to find a sensible name. If no common meaningful name can be found, the observed commonalities may be purely coincidental.

In the example of authoring messages in Eudora, it could be argued that it makes more sense to generalise `REPLY TO MESSAGE` and `FORWARD MESSAGE`, since they both require a source message, while `NEW MESSAGE` does not. However, we could introduce another generalisation named `REACT TO MESSAGE`, which would include a message resource in its definition. This illustrates another important point: Generalisations should have a purpose beyond noting similarities.

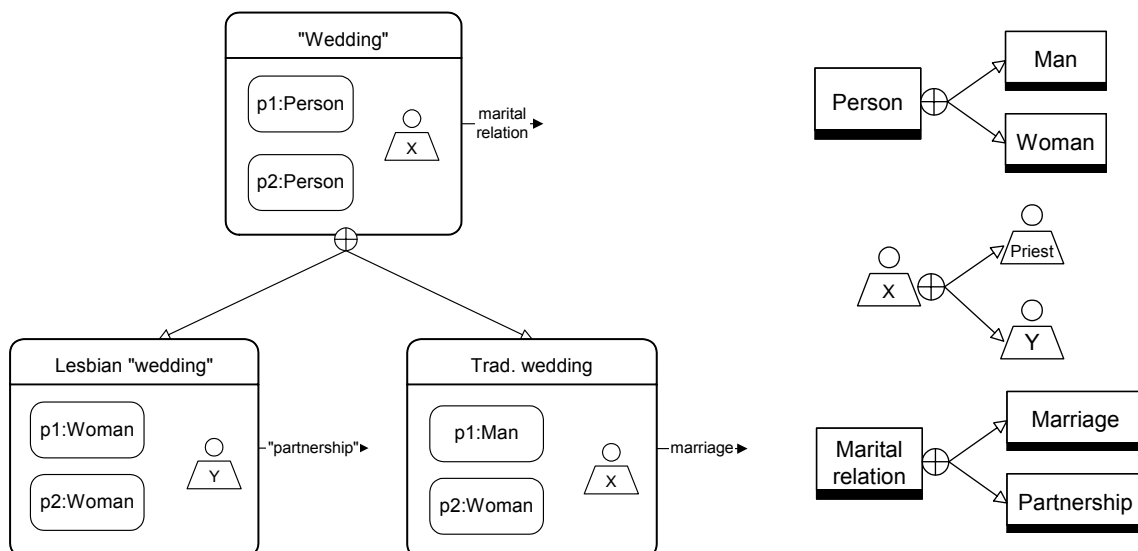
For the purpose of *design*, the similarities of several tasks identified in generalisation may give directions for a common design. I.e. if several tasks are classified as similar in a conceptual sense, it seems reasonable to have a similar design for them. For instance, the noted generalisation could be used to design a common interactor for the three authoring tasks, with variants for each specific one. This should make the design less complex, easier to learn and cheaper to make. From a *knowledge management* point of view, a generalisation can be viewed as improving the structure of a model. In an organisational context, it provides support for reuse of knowledge across projects. For instance, the two suggested generalisations above, makes the differences and commonalities among the three subtasks easier to note and understand. *Design patterns* represents a combination of the previous two, providing a certain approach to managing design knowledge. For instance, the idea that a set of specialisations could be supported by a common design, can be captured in a design pattern. Model-based design patterns will be discussed in Section 9.2.1.

A generalisation consists of two parts, the common characteristics identified and included in the general supertask, and the differing characteristics of the subtasks. The latter may further be divided in two: the distinguishing/differentiating characteristics which represents the reason for the differences, and the those that are the consequences of the distinction. For instance, a certain subtask may be optional in one subtask because a certain class of resources deems it unnecessary. The subtasks may specialise the general supertask in several ways:

- Sequence constraints and tighter post-conditions may be added, to reduce the number of allowed combinations of task performances. For instance, a set of non-overlapping subtasks (-) may be constrained to a specific sequence, using the sequence relation.
- Subtask cardinalities may be reduced, by shrinking the allowed interval. For instance, a task can be forced to be performed at least once by setting the lower bound, or in effect removed by zeroing the upper bound.
- Subtasks can be added and linked to existing task by introducing new sequence constraints. For instance, an extra validation task may in certain cases be performed, based on the costs of error.
- Additional resources may be required, or may be constrained to more specialised concepts. For instance, an additional resource may be needed to satisfy an added subtask, or a dynamic resource may be replaced by a constant.

The overriding principle is that the invariants of the supertask expressed as pre- and post-conditions should be preserved in the specialisation. Hence the constraints that are added in the specialisation must be consistent with the constraints that are already expressed in the supertask. Without consistency no actual task performance, i.e. task instance, could be classified as both being an instance of the general and special tasks at the same time. In addition, by adding the constraints, we ensure that any task classified as being an instance of the subtask will also be an instance of the supertask, but not vice versa.

Figure 50 shows another example of a specialised task, with pre- and post-conditions, in the form of two input elements, one actor resource and one output element i.e. post-condition. This particular model states that there are two different ways of wedding people, and that there are certain conditions that must be met for either of these. In the general case, two persons are wed by an actor (of a generic kind), resulting in the establishment of marital relation. If both the involved parties are women, the relation is called a partnership and the performing actor cannot be a priest. The other special task involves two of opposite sex, can in addition be performed by a priest, and results in a (traditional) marriage relation.<sup>1</sup> The accompanying domain and actor model fragments illustrate how a task specialisation often mirrors other specialisations, in this case, the input resource, actor and output elements. We can see that the two special tasks really are specialisation of the generic task, since they both require more constrained resources. That the output is more constrained too, is typical although not a necessity. The subtask structure is not involved in this particular specialisation, but it would be natural to include a “ceremony” task inside “Wedding”, with special variants in each branch of the main specialisation.



**Figure 50.** Task specialisation and specialisation of resources, actors and output / post-condition

Different kinds of characteristics of a set of tasks, may be the basis for generalisation. Since a generalisation consists of two part, the general and the specific, it can be classified in two ways, according the common characteristics gathered in the supertask, or based on the distinguishing characteristics found in each subtask. In our experience, the typical generalisation will focus on the common structure of a set of subtasks, and include extra sequence

1. The third case of gay “wedding” is left out for brevity, as we have no compact notation for expressing that two elements be the instance of the same concept, relative to a certain specialisation.

constraints in each subtask, based on characteristics of one or more resources. Based on observations of a set of users, one can model the personal preferences of a set of concrete actors. Using generalisations, a common action structure can be derived and provide a valuable understanding of more or less universal characteristics of a task. This is important for segmenting the user population and designing a common user interface for each segment. It can also be used for judging the need and potential for tailoring, i.e. to what extent each segment or user group should have a specialised user interface and how the variants of the user interface should differ.

## 4.8 Conclusion

We have presented a language for modelling tasks, TaskMODL, which is designed as a hybrid of traditional hierarchical task languages with sequencing constraints, and workflow-oriented languages with resources and dataflow. As a hybrid, TaskMODL provides concepts and notation from both worlds, making it a highly flexible notation. TaskMODL is tightly integrated with the RML domain modelling language, which is also used for defining the TaskMODL language. This makes TaskMODL an expressive language with a relatively simple interpretation.

In [Limbourg, 2000] several task modelling approaches and languages are analysed and compared. A common meta-model (henceforth shortened to CMM) is developed which includes the common concepts for hierarchical goal and task decomposition, temporal constraints, roles and objects and system actions (operating on objects). TaskMODL corresponds well with the common meta-model, with some differences:

- In CMM, goals form an explicit hierarchy with “accomplishing” relations into the task hierarchy. In TaskMODL, goals are part of the definition of a task, as post-conditions (and as pre-conditions of subsequent tasks), and are not themselves decomposed. A separate goal hierarchy may be built by collecting all explicit goals in the task hierarchy.
- In TaskMODL roles are modelled as actor classes in a hierarchy. Actual users or groups of such are instances of these classes. In CMM there is only a set of roles and no concrete users.
- The domain modelling part of CMM only includes an “object” concept, to express how tasks use and manipulate information. Our RML domain modelling language is much richer, although it does not include actions for manipulating the domain objects.

The main weakness of TaskMODL with respect to CMM is concerned with what [Limbourg, 2000] calls the operational level and operationalisation. This is due to our focus on expressing how tasks are performed without reference to a system, and RML’s lack of a data manipulation language. The DiaMODL language presented in Chapter 5, “Dialogue modelling”, will shift the focus to the user-system dialogue and include support for system actions.

## 4.8.1 RML and TaskMODL and the representation framework

RML and TaskMODL are designed for modelling the static and dynamic part of the task domain, respectively. In terms of the classification framework introduced in Chapter 3, they cover the problem-oriented perspective, by focusing on the task domain, rather than how a particular software solution support the tasks. RML is not a hierarchically structured language, e.g. there is no RML concept for domain and sub-domains, so the granularity dimension has no direct interpretation in RML. However, if we define concepts/sets as the objects of investigation, RML spans all levels from general concepts, via fine-grained concepts to specific instances. In addition, RML lets the modeller make precise and *formal* statements of the domain. TaskMODL is hierarchical and thus targets several levels of granularity. By basing the design on the APM workflow language, we believe it covers at least the bottom level of workflow. Since it contains the main elements of traditional task modelling languages, it should handle individual tasks down to atomic actions. TaskMODL includes constructs with varying formal strength, in the sense that more or less of the task domain is captured in precise and constraining statements. I.e. it is possible to formalise little or much by using weak constructs or more constraining ones, respectively.

Below we will discuss how RML and TaskMODL supports the 6 movements in the design representation space introduced in Section 3.6:

- Movements 1 and 2, along the perspective axis: From problem to solution and from solution to problem, respectively.
- Movements 3 and 4, along the granularity axis: From down the hierarchy from high-level descriptions to lower-level ones, and up the hierarchy from lower-level descriptions ones to higher -level ones, respectively.
- Movements 5 and 6, along the formality axis: From informal descriptions to more formal ones, and from formal descriptions to less formal ones, respectively.

For RML the relevant movements within the representation space are movements to and from design-oriented data-models, up and down in concept granularity and to and from informal natural language usage.

### Movements 1 and 2

RML has no particular support for moving to more design-oriented data models, such as implementing relations as attributes, defining traversal directions, ordering av sets or adding operations and methods. There is however a natural mapping to UML for most models. Going from data-models to RML, i.e. reengineering the domain concepts, requires stripping out design-oriented elements. RML provides no particular support for this.

### Movements 3 and 4

RML provides good support for defining finer-grained concepts, based on existing ones, through the two variants of the specialisation/subset construct. A special shorthand notation



is provided for defining concepts that require relation participation, like ‘Husband’ in Figure 25. Specialisation of relations is also supported. Several orthogonal specialisations may be defined, allowing different parallel classification hierarchies. It is similarly easy to define general concepts based on more special ones, whether normal concepts or relation concepts. In addition, individuals may be classified as belonging to a concept.

## Movements 5 and 6

The rich constructs for concept modelling mentioned in movement 3) are important when formalising a domain, e.g. based on natural language text. In addition, RML support annotating the model with variants of the same concept, like singular/plural for concepts/sets and different reading directions for relations. These annotations may also be used for generating natural language paraphrases of RML models.

The support of the six movements is fairly good, and as we will see, this support is inherited by the task modelling language, based on how it is defined in terms of RML.

For TaskMODL the relevant movements within the representation space are movements to and from models of dialogue and abstract interaction objects, up and down the task hierarchy, and to and from informal scenarios.

## Movements 1 and 2

TaskMODL provides the necessary constructs for moving to dialogue models. TaskMODL includes the information that is used by or in some sense is relevant for a task, and hence should be made available for the user through abstract interaction objects. The goals of a task can be made explicit, so that system functions can be designed for reaching those goals. TaskMODL also includes sequencing constraints, which are necessary for navigating between and activating and deactivating abstract interaction objects. Since TaskMODL is not designed for describing precise procedures, it does not necessarily provide enough information for automatically moving to or generating the abstract dialogue. It should similarly be possible to express in TaskMODL the possible task sequences for a particular dialogue, by using sequence constraints of varying strengths. The flexibility provided by non-strict trees of sequence constraints makes this easier than in other task modelling languages. This will of course depend on characteristics of the dialogue modelling language, but our Statechart-based approach described in Chapter 5, “Dialogue modelling”, provides similar constructs (parallel and disjoint composition) that fits fairly well with those of TaskMODL (aggregation and sequencing).

## Movements 3 and 4

TaskMODL provides good support for hierarchical modelling. Information present at one level, are available on the level below, and constraints expressed at one level, translate to constraints in a decomposition. The flexible notation and support for anonymous tasks, make it easy to add or remove levels in the hierarchy. The flexibility to include flows and constraints that span subtrees makes reasoning about task composition more complicated, however, as each task may interact with other than sibling tasks.

## Movements 5 and 6

The tight coupling between TaskMODL and RML provides good support for expressing the relation between tasks and the domain concepts, e.g. when formalising verb-noun statements. RML unfortunately lacks features for expressing more dynamic conditions within the domain, which may be necessary when formalising pre- and post-conditions for tasks. The varying strengths of the sequence constraints and the flexible constraint notation, provides good support for expressing weak and strong statements about how tasks are performed. Similarly, the support for classifying and specialising tasks, makes it possible to express general and specific statements about tasks, and conditions for when to perform variants of a task. Each construct can be translated to natural language statements about tasks. Complexity may be reduced by replacing several strong constraints with fewer weaker ones. Generating scenarios or task traces within the constraints expressed, is complicated by the flexible constraint notation.

The support of the six movements is in general fairly good. The flexibility of the constructs and notation provides good support for formalising the task domain, but on the downside it makes reasoning more difficult.

# Chapter 5

## Dialogue modelling

In Chapter 2, “State of the Art”, we suggested three main user interface design perspectives, the task/domain perspective, the abstract dialogue perspective and the concrete interaction perspective. The former was described as a problem-oriented perspective, while the latter two were described as solution-oriented. In this chapter we focus on dialogue, and its relation with task and concrete interaction, i.e. the two neighbouring perspectives. In our view, dialogue modelling is concerned with dataflow, activation and sequencing, while concrete interaction focuses on the dialogue’s relation to input and output devices, e.g. windows, widgets and mice. Although we distinguish between these two perspectives, they are partly integrated in one modelling language.

### 5.1 Introduction

In the following sections we will present the DiaMODL language which is used for modelling dialogue, and later some aspects of concrete interaction. The language is based on the Pisa interactor abstraction as presented in [Markopoulos, 1997]. The interactor abstraction was chosen because it has been revised several times and is a mature abstraction [Duke, 1994]. More precisely, the interactor abstraction presented in this chapter is close to the extensional (black-box) description of Pisa interactors in [Markopoulos, 1997], but our interactors do not have the same white-box structure. In addition, we use Statecharts instead of LOTOS for formalising the behaviour. This choice is based on our two-fold goal: 1) defining a language that can be used as a practical design tool, and 2) make it easier to build runtime support for executing interactor-based interfaces, e.g. through the use of UML. While we appreciate LOTOS’ support for machine analysis, we feel the simplicity and the look & feel of Statecharts makes it a better candidate for our purpose. In addition, the correspondence between Statecharts’ and/or decomposition and RML’s aggregation/disjoint subsets, can be utilised for combining static and behavioural modelling of interaction, as indicated in [Tr etteberg, 1998]. Finally, since Statecharts is part of the UML standard, we believe an implementation based on Statecharts is more realistic and practical for industrial use.

However, the most important aspect of interactors is not their formalisation in terms of either LOTOS or Statecharts, but rather how it lets the designer decompose the user interface into similarly looking components, or alternatively, compose a user interface from

components. These components have the same generic structure, but can be tailored to have different functionality within a very large behavioural space. This is the main reason for using the interactor abstraction in the first place. In addition, interactors provide an abstract view of the concrete interaction objects found in existing toolkits, and hence can be used for abstract design and analysis of interface structure and behaviour.

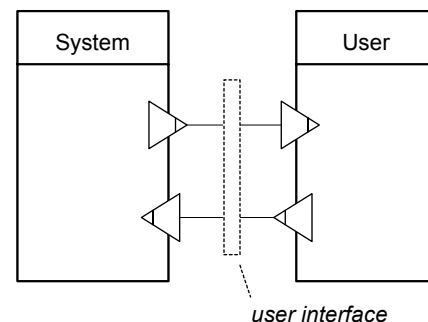
Different types of interaction objects have different capabilities and each type will typically be oriented towards a particular role in the implementation of the abstract dialogue. We have identified three roles: information mediation and flow, control and activation and compositional structure. These roles correspond to the three main parts of an interactor's function and behaviour:

- *Information mediation*: The user interface can be seen as a mediator of information between a system and its user, as well as a tool for performing actions. Each interactor defines the information it can mediate and in which direction the information flows.
- *Control and activation*: Interactors have an internal state space controlling its behaviour, which includes sending and receiving information, activating other interactors and triggering the performance of application specific functions.
- *Compositional structure*: Interactors can be composed into new interactors, to make it possible to build higher-level behaviour and abstract away details.

In addition to these three aspects of the interactors' definition, interactors can be parameterized to provide tailorable behaviour that is suitable for a large range of applications.

## 5.2 Interactors as information mediators

A user interface can be seen as communication channel between a system and a user. As illustrated in Figure 51, the information can pass through the channel in either direction. Information passing from the system to the user will typically indicate the system's state, including the actions the user is allowed to take. Information passing from the user to the system will typically be actions with attached data, which is meant to modify the system's state. The pattern of information flow depends on the dialogue style of the application, but will typically alternate between system-user and user-system.



**Figure 51.** System-user communication

A dialogue model should describe the structure and behaviour of the user interface, so we need to be more explicit about this communication channel. In an alternative view illustrated in Figure 52, the user interface is given a more active role as that of a *mediator* of information. The system can output information to the user, by sending it out of the system into the user interface, which in turn may pass it on to the user in some form suitable for

both the user and the target platform. The user can input information to the system, by sending it into the user interface, which may pass it on in a form more suitable for the system.

The information flowing to and from the system will be based in the task model, since it captures the domain the user interface provides access to. However, the information flowing between the user interface and the user must be based on the target platform's devices, e.g. the output may be text, graphics and pictures, and the input may be keypresses and mouse clicks. Since these are outside the scope of an abstract model, we leave out from our model, the terminal flow of information between the user interface and the user, as indicated in Figure 53. We will still model information flow to and from *parts of* the user interface in the direction of the user, since this is part of the structure and behaviour we want to capture in an abstract dialogue model.

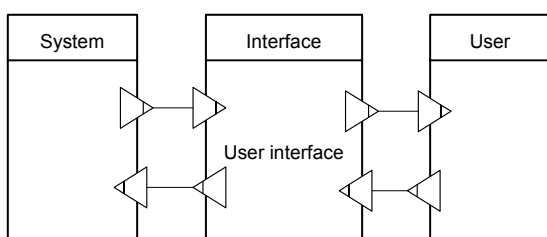


Figure 52. Mediation between system and user

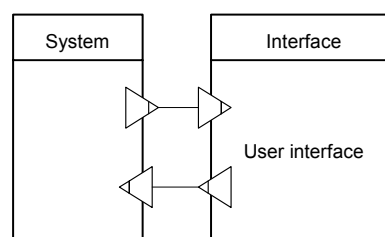


Figure 53. User interface only

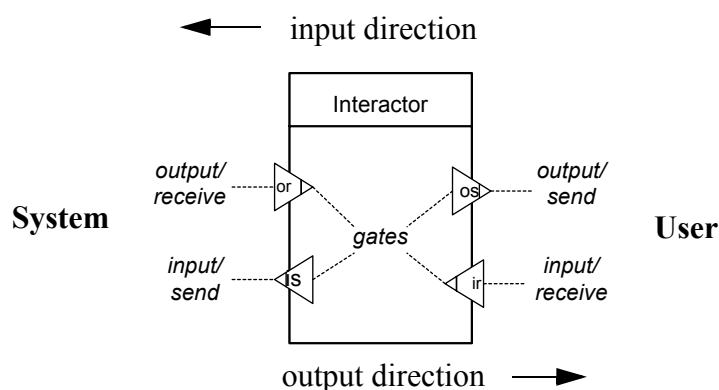


Figure 54. Generic interactor

The *interactor* concept can be seen as such a generic mediator of information, between the system and user, in both direction, or the generic component from which such a mediator is built. I.e. an interactor mediates both input from the user to the system as well as output from system to the user, as illustrated in Figure 53.<sup>1</sup>

The information mediation aspect of interactors and structures of interactors is defined by

1. a set of gates defining the interactor's external interface to the system, user and other interactors,

1. In all the figures the system side is to the left and the user side is to the right. These correspond to the abstraction and display side in the ADU process in [Markopoulos, 1997].

2. a set of connections that carry data between gates, and
3. the control structure that trigger or inhibit dataflow between gates on the interactor's in and outside and along connections.

Together these provide dataflow between interactors, synchronization and activation of interactors.

### 5.2.1 Gates, connections and functions

An interactor receives and sends information through a set of *gates*, i.e. the gate is the mechanism by which information flows into and out of interactors. Each gate consists of two parts, the *base* and the *tip*, each holding a *value*. The gate *receives* values in the base and *sends* values from the tip. The values in the base and tip may differ, and in particular, the tip's value may be computed from the base's value. Both the base and tip are *typed*, and are constrained to only hold values of that type. Each gate is attached to one interactor, and the base and tip will be on different sides of the interactor's boundary. As can be seen in Figure 54, there are 2x2 kinds of gates, for sending and receiving in each of the two directions. Their role and names<sup>1</sup> are as follows:

1. *Input/send (is)*: Input originating from the user's interaction with input devices, results in information flow towards the system, out of the interactor.
2. *Output/receive (or)*: Output information flow towards the user, into an interactor responsible for outputting it to the user.
3. *Input/receive (ir)*: Information in the direction from the user to the system, flows into the interactor for further processing/meditation. As discussed above, information from devices will be left out of the model.
4. *Output/send (os)*: Information in the direction from the system to the user, flows out of the interactor for further processing/meditation by other interactors. As discussed above, information to devices will be left out.

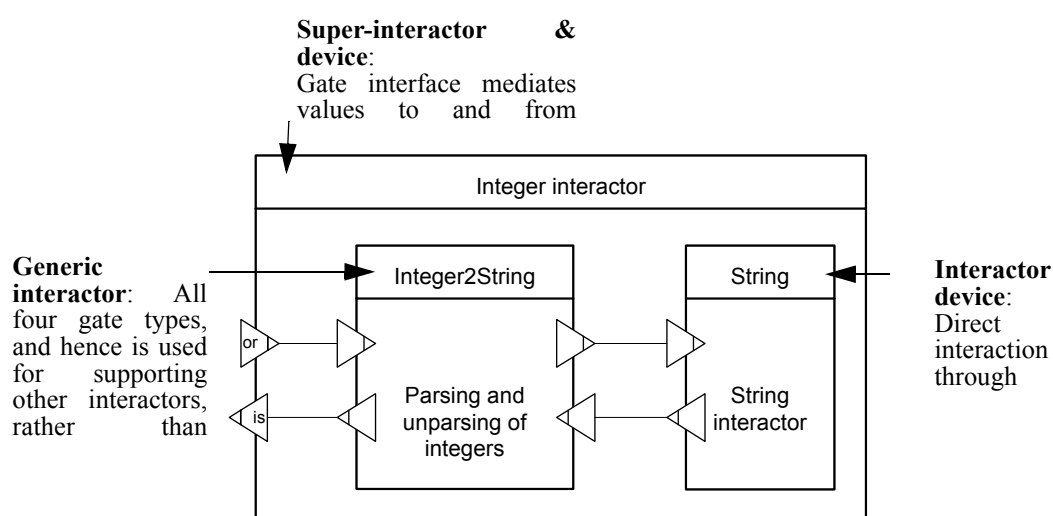
Gates may be labelled by the abbreviation (“is”, “or”, “ir” or “os”), to make the models easier to interpret. The inner workings of a gate will be detailed in Section 5.6.4, for now it suffices to say that its function may be monitored and controlled by connections to its side. The base and tip are normally on opposite sides of the boundaries of an interactor.<sup>2</sup> For instance, an input/send gate will have the tip on the outside and the base on the inside of the hosting interactor, while the opposite is the case for output/receive gates. Most interactors will have only input/send and output/receive gates, like those that correspond to elementary concrete interaction objects or compositions of them. In fact, few interactors will have all the four kinds of gates, like the generic interactor in Figure 54 has. The difference and usage of various interactors becomes apparent when we look at composition of interactors.

Figure 55 shows an interactor for the output and input of integers, implemented as a composition of a string interactor and an integer-string/string-integer converter. As can be seen, the

1. Their names are similar to the ones of the Pisa interactor, as presented in [Markopoulos, 1997].

2. The base and tip are graphically separated by a line, which sometimes looks as if it *is* the interactor boundary.

interactor abstraction is compositional, i.e. the composition is itself an interactor, and from the outside, the INTEGER INTERACTOR is just another interactor. Both the inner STRING interactor and the composite INTEGER INTERACTOR are examples of what we term interactor *devices*; interactors used for the output and input of one datatype. Such devices have gates only on the left/system side, i.e. input/send and output/receive gates, since they interact *directly* with the user through platform devices (physical or window system objects) and not through other interactors. Pure input devices like mice have only input/send gates, while pure output devices only have output/receive gates. The INTEGER2STRING interactor works as an adaptor, by mediating/converting values in both directions: In the output direction, the conversion increases the semantic content or understandability of data for the user, while decreasing it for the system. For instance, the character string “123” is more understandable to the user than the 111011 bit pattern corresponding to the 123 integer. In the input direction the semantic content increases for the system, while it decreases for the user. For instance, “123” is converted to the integer 123 represented by the 111011 bit pattern.



The connections inside the INTEGER INTERACTOR and the control logic within INTEGER2STRING ensure that values are mediated from the outer output/receive gate STRING’s output/receive gate, and converted to a string that is presented to the user. Similarly, the string input by the user through interaction with STRING is converted and mediated to the outer input/send gate. Note that the nature of interaction provided by STRING and the language supported by INTEGER2STRING is not represented in this particular model. For instance, STRING INTERACTOR may be a simple text field, a full word processor or a speech synthesizer and analyser. Similarly, INTEGER2STRING may support Arabic or Roman numerals, or natural language numbers in Norwegian or Sanskrit. Since the base and tip of gates are typed, the connections between the gates can be verified to ensure consistency at a syntactic level. This will typically be checked by model editing tools, when the gates of interactors with known type signature are connected. For instance, it can be checked that the connections between the INTEGER2STRING and STRING interactors are valid. Alternatively, connections may constrain the type of gates that do not yet have a type. For instance, the gates in the left hand side of the INTEGER and INTEGER2STRING interactors must have the same type, since they are directly connected, although the actual type may not be defined until later.

Based on our prototyping experience, we have added a type to connections, to filter out values. For instance, if the input/send gate of the INTEGER integer should only emit values below 10, the connection at its base could include a type specification that constrains the values it may mediate to such integers. A set of connections with disjoint filters may thus act as a kind of case switch.<sup>1</sup>

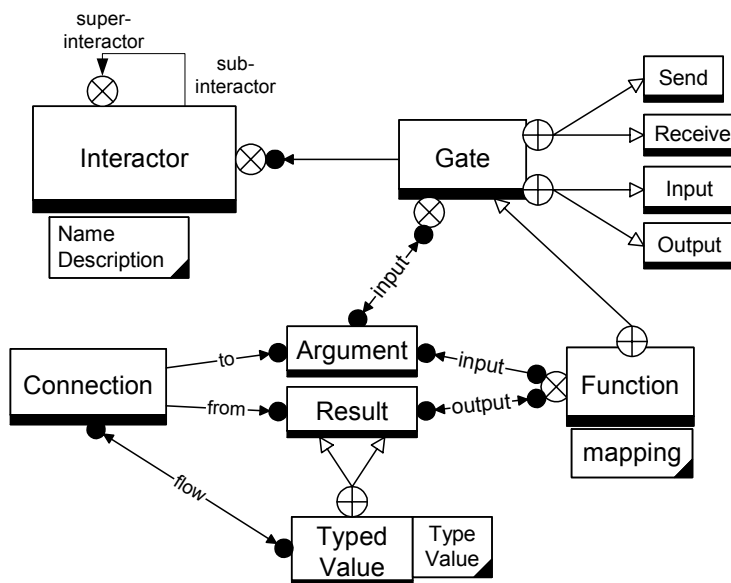


Figure 56. Interactor concepts

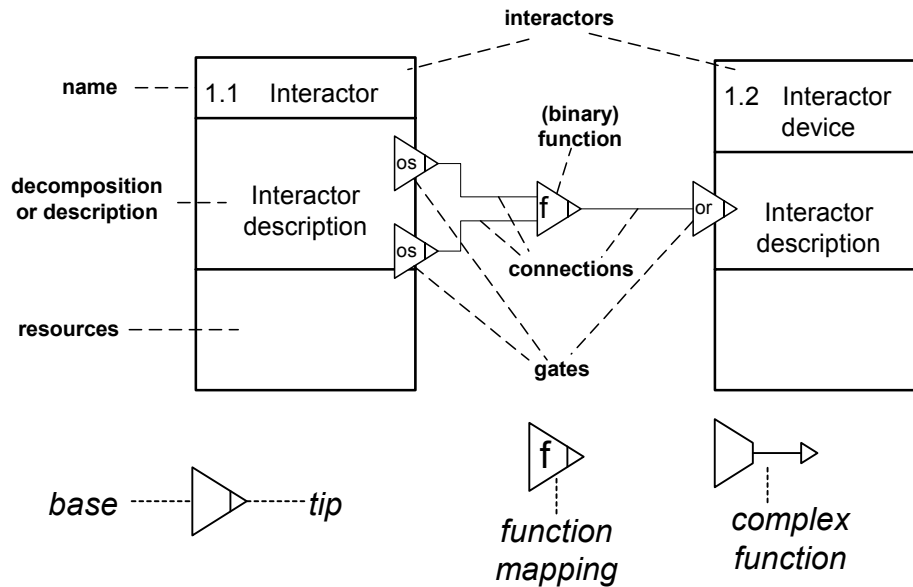
The concepts introduced so far may be modelled in RML as shown in Figure 56, and the notation is summarised in Figure 57. The INTERACTOR is an aggregation of a set of GATES, each of which must belong to one and only one INTERACTOR. Each GATE is classified in two different ways, giving the four classes listed above. A (SUPER-)INTERACTOR can further contain other (SUB-)INTERACTORS. The interactor symbol is a rectangle with up to three compartments: the name (mandatory), the textual description or decomposition and its resources. Figure 55 exemplifies the notation for interactor decomposition, while resources will be discussed in Section 5.5.

As can be seen from the model, the GATE concept is based on an underlying FUNCTION concept, which takes a set of input ARGUMENTS and computes an output value RESULT, based on a function mapping. The two-part triangle symbol is used for FUNCTIONS, too, labelled by the mapping (name). The input values are visually represented by the base and the output value by the tip. A GATE differs from a FUNCTION by only having one input value, and by mediating values across an interactor's boundary, according to the 2x2 classification. The CONNECTION concept is shown to be related to the RESULT from a FUNCTION and a function ARGUMENT, indirectly connecting two FUNCTIONS (and/or gates).<sup>2</sup> In addition, a CONNECTION provides support for mediation and filtering of one value (at a time), by being related to a TYPED VALUE of its own. A connection is constrained to only connect functions within or at the boundary of the same interactor, hence a connection may not cross an interactor's

1. This will be similar to the XOR-splitter construct of APM, described in figure 5.17, p. 130 in [Carlsen, 1997]

2. Since there is no visual indication of a separate connection point for each input value on a base, it may be difficult to differentiate between several connections to the same input value, and separate connections to different values.





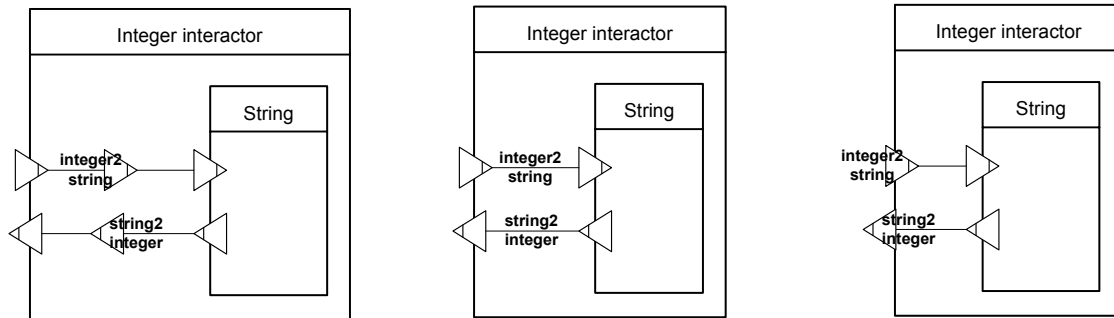
**Figure 57.** Summary of the notation for interactors, connections and functions

boundary. This ensures that the interactor's gate signature fully captures the information flowing into and out of the interactor.

The need for including generic functions is apparent, as the structure of connected gates provides information flow from user to system and back, but does not provide a way of computing new values. For that we need access to domain or application specific functionality, like the integer parsing and unparsing functions which are needed by the `INTEGER2STRING` interactor. By modelling gates as special functions, we get a more compact and flexible notation, particularly since gates often can make good use of the function mapping (which defaults to the identity function).

Figure 58 shows three ways of implementing the `INTEGER` interactor using functions. In the left variant, standalone functions are inserted to perform the necessary parsing and unparsing of strings. To make the notation less verbose and more practical, connections can be labelled with unary functions instead of inserting the function symbol. This is shown in the middle variant. Finally, functions can be used at the boundary of interactors as special gates, as shown in the right variant. I.e. the left model in Figure 58 can be simplified by either 1) removing the function symbols and adding them to the connection, as in the middle, or 2) replacing the two output/receive and input/send gates with the `integer2string` and `string2integer` function gates, respectively, as on the right. As a final notational convenience, a stippled connection is interpreted as the complement function, typically used for inverting boolean values.

Although a function may be considered atomic for some purposes, it may nevertheless be complex, run through a series of steps, require interaction and take considerable time to compute. To indicate that a function may have a more complex behaviour than instantaneous computation of a value, a variant of the function symbol is used, where the base and tip are ripped apart, as shown bottom right in Figure 57. Note that if the result is not used (locally), the ripped off tip may be left out. Complex functions will be further discussed in Section 5.6.



**Figure 58.** Functions: standalone (left), on connections (middle) and on gates (right).

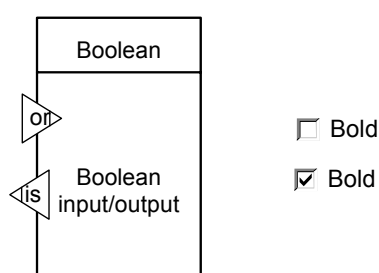
By comparing the signature of the function with the types of the connected gates, some level of consistency can be ensured. However, an appropriate domain language is needed for expressing (part of) the semantics of the functions. We will be using RML as our domain language, and this will allow us to express mappings from instances to attribute values, mappings between instances using (derived) relations and filtering of sets. Such functions can be automatically defined from an RML model, and hence be made available for use in our interactor-based dialogue models. The following are useful functions that should be automatically defined:

- For each concept, a predicate  $\langle concept \rangle?(element) \Rightarrow boolean$  for testing whether an element is a member of the concept's extension.
- For each concept/set, a function  $\langle concept \rangle(set) \Rightarrow set\ of\ \langle concept \rangle$  for extracting elements in that class. This is particularly useful for extracting one of a set of disjoint subsets of a general concept, e.g. the “men” subset of “persons”.
- For each concept, a function  $make-\langle concept \rangle(...) \Rightarrow element\ of\ \langle concept \rangle$  for creating new element for a concept. The argument list may either be empty, or could at least include all required attributes and the elements needed for establishing the required relations. For relation concepts, the argument list must include elements for each end.
- For each concept, a function  $change-into-\langle concept \rangle(element\ of\ \langle concept \rangle, ...) \Rightarrow element$  for re-classifying an element to that concept. The argument list may either be empty, or could at least include all required attributes and the elements needed for establishing the required relations.
- For each attribute of a concept, two functions  $\langle concept \rangle-\langle attribute \rangle(element) \Rightarrow \langle attribute\ type \rangle$  and  $set-\langle concept \rangle-\langle attribute \rangle(element\ of\ \langle concept \rangle, \langle attribute\ type \rangle)$  for returning and setting that attribute of an element.
- For each relation, two functions for mapping from each end to the other. The signature will depend on the cardinality, but will in general be  $(element) \Rightarrow set$ . Their names can be based on either the relation name, the relation and concept names, or the (role) name defined for each end.
- For each relation that has incomplete coverage, a predicate for testing whether an element participates in the relation.

## 5.2.2 Basic device interactors

Most concrete interaction objects are interactor devices, i.e. they are used for input and output of specific data types with direct interaction with the user. In this section we will present the basic interactors and the various ways they can be implemented in a standard toolkit. Although these interactors may be decomposed into a set of controlling states, they are atomic with respect to input and output of values.

The simplest interactor device is for boolean input and output and the prototypical toolkit equivalent is the checkbox. As shown in Figure 59, the checkbox has two states corresponding to the two possible values, false and true, usually indicated by a check mark or cross. The checkbox label is used for relating the boolean value to its interpretation with respect to the underlying domain model, and does not affect the checkbox behavior.

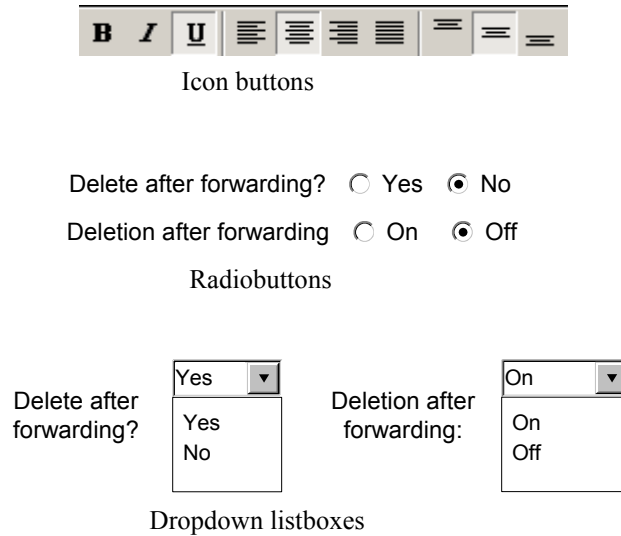


**Figure 59.** Boolean interactor device (left) and the corresponding concrete interaction object (right)

The typical usage of the boolean interactor is for classifying an instance, or for editing the value of a boolean instance attribute. It may also be used as an action modifier by selecting between variants of the same action, corresponding to (task) instance classification.

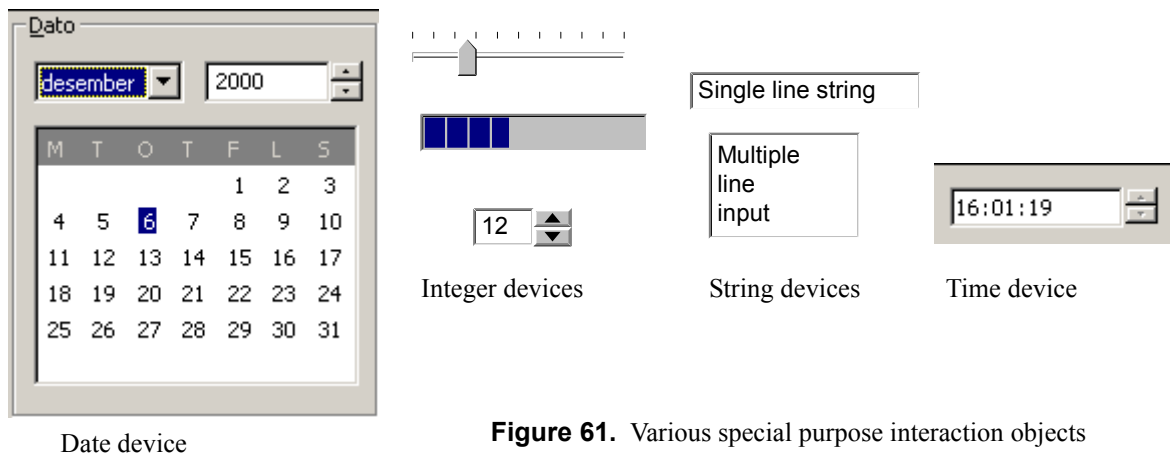
The same function as a checkbox, can be achieved using an icon button, where the pressed and released states indicate true or false, respectively. The button has the same toggling behaviour as the checkbox. In Figure 60, the three (independent) character attributes **bold**, *italic* and underline are controlled using icon buttons, each button implementing a single boolean interactor. The main advantage compared to checkboxes, is that they require less space and the icon may be easier to recognize than the label. A quite different solution is using a pair of radiobuttons or a pulldown listbox with two elements, as a special case of selecting one of a set of (mutually exclusive) options. For both, the two states are indicated using two different wordings with opposite meaning. This is typical example of how the same abstract function, in this case the model in Figure 59, may be implemented using different concrete interaction objects. Which concrete interaction object to actually use depends on a number of factors, like user's mental model of the domain (is true/false more intuitive than yes/no or on/off), user capabilities (do not use red/green lights for indicating stop/go if they are colour-blind), and space requirements (checkboxes require less space than radiobuttons).

All these interaction objects use screen space and mouse clicking as main resources. An interesting variant is a time based one, where a(ny) particular event triggered by the appropriate device in a certain time interval, is interpreted as true, with no event corresponding to false. This is portable to any device or modality, and illustrates the extent to which concrete implementation of the abstract interactor definition can vary.



**Figure 60.** Other concrete interaction objects implementing the boolean interaction device

The icon button is an example of a concrete interaction object that may be used for implementing several abstract behaviours. For instance, two-state icon buttons can be grouped and controlled so only one is down at a time, to implement selection among a fixed and limited set of choices. The seven icon buttons at the top right in Figure 60 behave as two groups of four and three, and are used for setting horizontal and vertical paragraph alignment, respectively. Both alignment group work as a unit, implementing a selection interactor, as modelled in Figure 62.

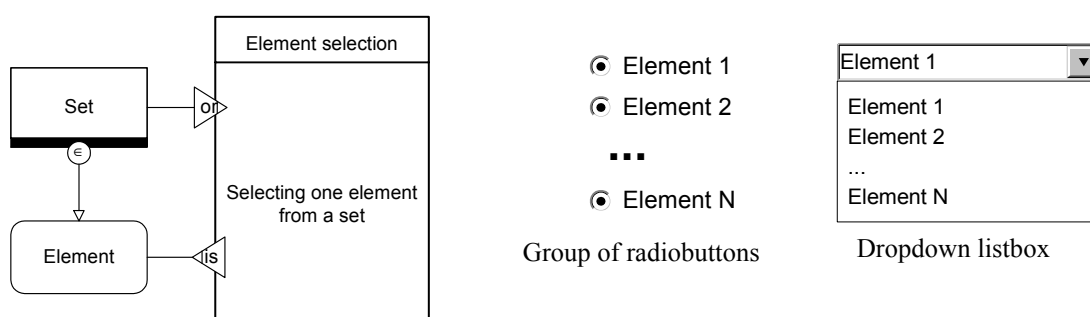


**Figure 61.** Various special purpose interaction objects

The string device interactor can be used to implement interactors for almost any basic data type, like integer, real, time/date etc., provided suitable parsers and unparsers. In addition, there exists many special purpose interactors for these types, some of which are shown in Figure 61. The date device is another example of an interactor built from simpler ones, like number range (year and day) and option selection (month) devices. In principle, each part could be replaced by functionally equivalent interactors, without changing the composite interactors definition, e.g. radiobuttons for the months, a slider for the day and even a calculator or spreadsheet for the year.

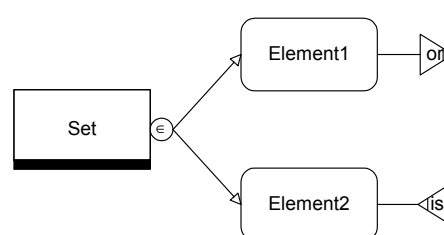
### 5.2.3 Set-oriented device interactors

The general function of groups of radiobuttons and (dropdown) listboxes is element selection, the interactor definition of which is shown left in Figure 62. The output to the user (output/receive of the interactor) is a set, and the input to the system (input/send) is an(y) element of this set. The manner of selection is unspecified, and will depend on the look & feel of the concrete interaction objects, e.g. radiobuttons and listboxes, as shown right in Figure 62. As noted above, any set of boolean interactor devices can be encapsulated to implement element selection, by ensuring that exactly one boolean device represents the true value at a time, and this is essentially the function of (groups of) radiobuttons. The prototypical element selection interaction object is the dropdown listbox, shown in Figure 62 in the dropped state. Again the choice of concrete dialogue for the abstract interactor depends on the circumstances, like the space constraints, the number of elements in the set, and whether the set is static or dynamic. In the standard case, where an element of a concept's extension is to be selected, the listbox is a natural choice, since it provides good support for long and dynamic lists. Many toolkits support typing for quickly locating elements with a specific prefix, and can be used for very large sets. For short and static sets, a group of radiobuttons is a reasonable alternative, the main advantage being that all elements are directly visible. In many cases, a table with many columns and potentially non-string fields, may be used for element selection, especially when its elements are used as the primary object of a task.



**Figure 62.** Element selection interactor (left) and two corresponding concrete interaction objects (right).

It can be argued that text fields can be used for element selection, too, especially if the text field is aware of the set of options, and provides completion. However, there is actually a subtle difference between this case and the one modelled by Figure 62. Since a text field does not directly show the whole set, but only the current selection, we should use the model shown in Figure 63, instead. We see that the output value in this case is a single element, as is the input value. Similarly, input-only interaction can be achieved by *removing* the output/receive gate. This may be relevant for input of familiar domain values using input-only devices. For instance, when inputting your birth date using speech, there is really no need to spell out the alternatives, as the date interactor shown in Figure 61 does. The need for explicitly modelling such nuances, will vary and may be a matter of taste. The model in Figure 62 may work well as an idiom, and the model in Figure 63 may be used in the case where such a distinction matters.



**Figure 63.** Alternative selection interactor

A less common selection task is multiple selection, an interactor for which is defined in Figure 64. The prototypical interaction object is the listbox, shown in Figure 65.

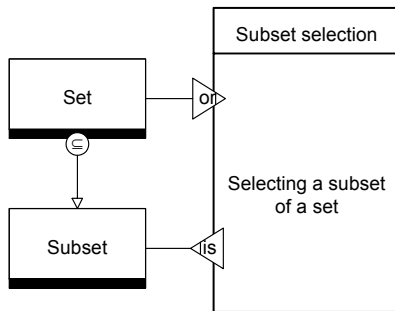


Figure 64. Subset selection interactor

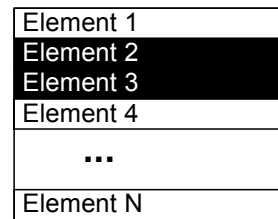


Figure 65. Listbox subset selection

It is perhaps surprising that subset selection is far less common than element selection, since many functions can be defined to take set arguments, e.g. object deletion, copying and printing. However, it may be confusing if only some functions have this feature, so subset selection is often deliberately avoided. As for element selection, the subsets can include (sets of) sets or categories, which is useful in search applications. While most list oriented interaction objects can be configured to support multiple selection, the dropdown listbox used for single element selection cannot function as a subset selection interactor. Similarly, groups of radiobuttons must be replaced by checkboxes if the abstract requirements change from element to subset selection.

More complex domain model structures require more complex interactors. The hierarchy is a very common structure, an interactor for which is shown in Figure 66. Using RML we can define a hierarchy in terms of one generic node concept, and two more specific ones, for interior and leaf nodes, the difference being that the former contains other nodes, while the latter does not. The output/receive gate of the interactor receives a set of interior containers, which are presented for the user. One of the leaf nodes in the hierarchy may be selected and emitted from the input/send gate of the interactor.

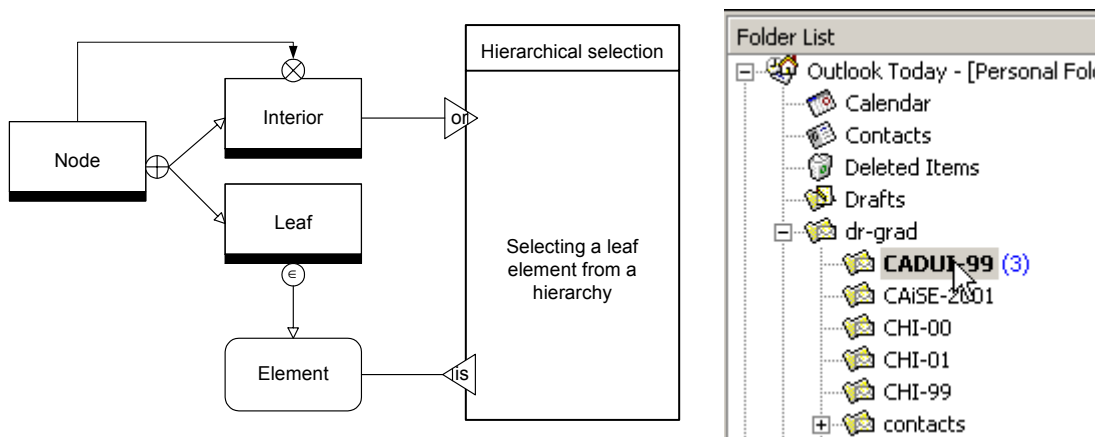


Figure 66. Selecting a leaf element from a hierarchy

The prototypical interaction object for hierarchical selection is the twist down tree (Mac Finder) and the folder view (MS Windows), as shown at the right of Figure 66. Different variants exist, and most toolkits lets the designer control if interior nodes also can be selected, corresponding to attaching the element symbol in the model to the INTERIOR sym-

bol instead of the LEAF symbol. The folder list shown in the figure, may present the whole hierarchy to the user, while other hierarchy interactors show only one level at a time and let the user drill down and back up. One way of capturing this difference is by adding an output/receive gate for the aggregation relation itself, in the case of the folder list. This highlights the fact that both the set of elements and their container/part relations are output to the user. As for the case of selection elements from sets, the need for modelling this difference may vary.

For information mediation, the power of the dialogue modelling language depends on the data modelling language used. The range of data values that may be output or input through an interactor, is not limited to the constructs of RML. For instance, the concept of state will be used to model how functions compute values, and this state is something the user should be aware of, and hence could be an input to an interactor. Similarly, data arrays and indices are data modelling concepts that RML lack, that are natural values to output to the user and receive as input. UML, with its focus on modelling software rather than the concepts of a domain, is a good candidate for the data modelling counterpart of dialogue modelling language. This is particularly true if UML is used for data modelling within the system or application perspective. For the same reason, Java or the CORBA's Interface Description Language (IDL) are probable data modelling languages.

## 5.3 Interactor content

In the previous section the interactor definition of a set of standard interaction objects were presented. Each interactor can be viewed as the specification of a certain input and output functionality, and the concrete interaction objects are the implementation of this functionality. In the case where a suitable interaction object exists, the implementation reduces to the mapping from the abstract interactor to the concrete interaction object. In cases where no corresponding interaction object exists, there will be a need for specifying the behaviour in more detail, and composing more complex interactors from simpler ones, or making a concrete implementation according to this specification from scratch. To understand how such composition is performed, we will in this section discuss the control and composition aspect of interactors.

### 5.3.1 Interactor control

The control aspect is concerned with triggering of information flow and activation and deactivation of interactors. In our interactor language, we have chosen to use the Statecharts language for this purpose. As shown in Figure 67, each interactor can be considered a Statechart super-state, which is entered when the interactor is activated. Alternatively, when an interactor is entered, it is considered active, and may respond to information propagated into its input/receive and output/receive gates and may emit data through its input/send and output/send gates.

The interactor may be decomposed into a hierarchy of substates, according to the Statecharts language. The main Statechart constructs are exemplified in Figure 68. The left part

of the figure, shows our notation for an interactor decomposed into a hierarchy of substates. The right part shows a tree view of the same hierarchy.

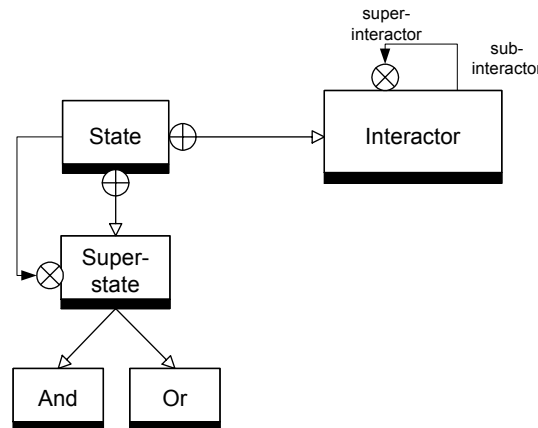


Figure 67. State-oriented interactor concepts

The interactor is decomposed into two “compartments” divided by a stippled line, to indicate and-decomposition. In the tree view, this is indicated by the “and” label by the top s0 node. Each compartment is in fact a substate of the top-level interactor, as shown in the tree view. Because they are and-composed they will both be active at the same time. The left substate compartment is further or-decomposed into states s1, s2 and s3. These substates are drawn using the traditional round state symbol and are connected with transitions, shown as arrowed lines. Or-decomposition implies that one and only one of the substates is active at any point in time, including initially, when the or-decomposed super-state is entered.

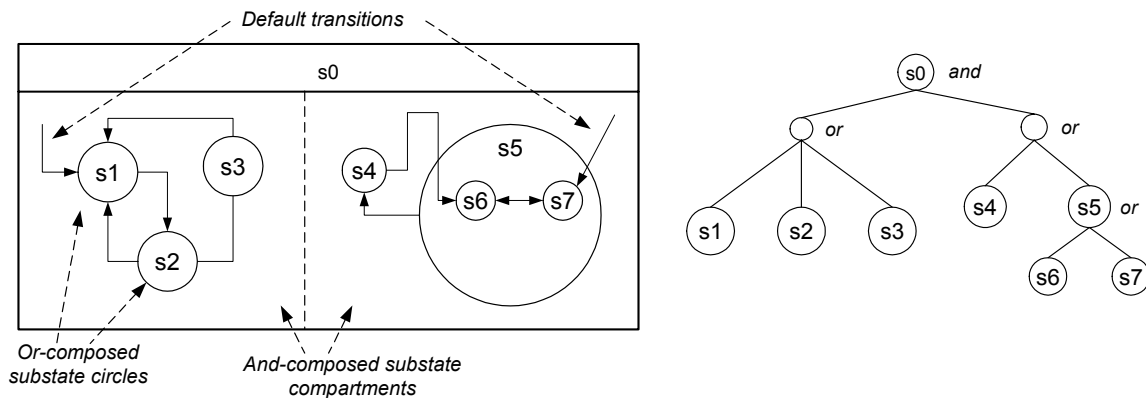


Figure 68. Main Statechart constructs.

- Left: our notation.
- Right: corresponding state hierarchy

Transitions lead from one state to another, and when it is triggered and followed, the state machine will exit the from-state exit and enter the to-state. Transitions may be labelled with events that triggers it, a condition that controls if it can be triggered and actions that are performed when the transition is followed. The notation used is *event[condition]/action*. If no event is given the transition always triggers when the condition becomes true. The condition defaults to *true*, so if no condition is given, the event will trigger the transition by itself. The action is either an event that may trigger other transitions in the state machine, or a call to an

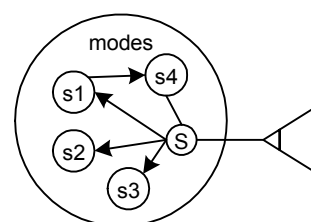


application specific function. Note that a transition may cross state boundaries and may only lead to a substate of an or-composition.

States may be referenced in both the event and condition part, and the traditional dot-notation will be used to refer to substates. The event corresponding to a state is emitted when the state is entered. When a state is used as a (predicate in a) condition, it refers to whether the state is active or not. Gates may similarly be used as events and conditions. The event corresponding to a gate is emitted when it either receives or sends a value, depending in whether it is a /receive or /send gate, respectively. When a gate is used as a (predicate in a) condition, it refers to whether it holds a value or not.

If a transition leads to (and not into) a state that is or-decomposed, a *default* substate is entered by means of a default transition with no originating state. In the example in Figure 68, both the left and right parts of the s0 interactor is automatically entered when s0 is activated. The left default substate s1 is then entered through the default transition attached at the left of s1. The right substate compartment is or-decomposed into two states s4 and s5, the latter of which is further or-decomposed into s6 and s7. The default transition in the right compartment leads to s7 within its superstate s7, which implies entering s5 first. There are two transition that will result in exiting s7, the one leading to s6 and the one leading from s5 to s4. If the transition from s5 to s4 is followed, both s7 and s5 will be exited, before s4 is entered.

A special *selection* transition support explicit selection of the target state of a transition. The transition leads to several potential target states, and the input to the special selection symbol, an encircled S, specified which target state to actually enter when the triggering event happens. Figure 69 shows an or-decomposed MODES state, with four sub-states s1 to s4. A selection transition leads from s4 to s1, s2 and s3. Which of these three states to actually enter is determined by the value emitted by the function at the right. This mechanism is useful in cases where a user interface element is directly linked to the state of an object, e.g. the minimized/normal/maximized state of a window frame.



**Figure 69.** Selection transition

The checkbox in Figure 59 has a behaviour that may be modelled by decomposing it into a set of states. By operating it and observing the appearance and behaviour, we can deduce the Statechart model shown in Figure 70. The main features are: 1) the gates representing the input and output values at the left, 2) the control state in the middle, and 3) the 2-part visual appearance at the right. These three parts are and-composed, i.e. operate in parallel and coordinate/synchronise through events and conditions. The gates part is mandatory for all interactors, while the two other parts are specific for this particular model. It is however natural to isolate the main logic from the visual appearance, as has been done in this model.

The GATES part defines how input and output values are propagated or synchronized, in this case the checkbox input/send value is connected to and hence equal to its output/receive value. If the toggle action occurs, a new input value is computed by a complement connection (the stippled line indicates an inverter function). The toggle action originates in the CONTROL substate, when a mouse button is pressed (transition from 1 to 2) and subsequently released (transition from 2 to 1). Note that the pointer must be INSIDE the checkbox

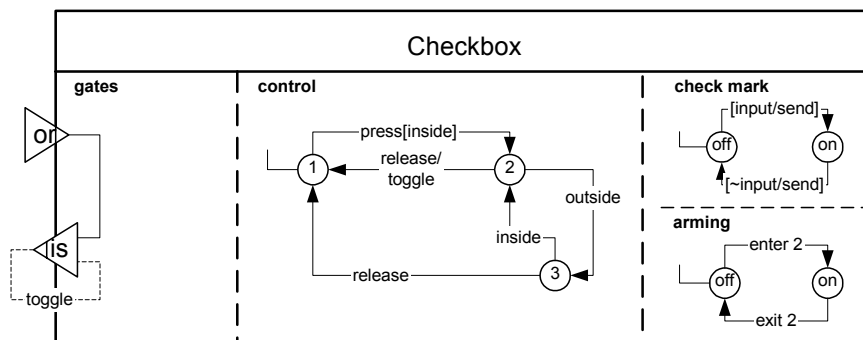


Figure 70. The checkbox behaviour

for both the press and release, but may be OUTSIDE in-between. In general, a connection may be labelled by both an event and a condition that states at what point in time it will propagate or synchronise a value. Events are used to trigger propagation, while conditions are used to limit when propagation may occur. The visual appearance is controlled through the CHECKMARK and ARMING substates, where the latter is used to indicate to the user if the toggle action will occur upon releasing the mouse button, i.e. the difference between states 2 and 3. The model can be simplified by removing the pair of ON/OFF states and inserting corresponding transition actions in the CONTROL part, for controlling the visual appearance, instead. However, this model makes the three parts, gates, control and appearance more explicit.

Two aspects of this state machine is left implicit: The meaning of the external events and conditions, like press, release and inside, and the correspondence between two pairs of on/off states and the visual rendering of the interaction object. I.e. where do the events originate, which external states and functions can be used in conditions, and how may external output devices be controlled? To make the model explicit and executable, we would have to link the state machine to the mouse device and window rendering function, after which this can be considered a complete implementation of the desired boolean input/output device. This will require interactor resources, which will be discussed in Section 5.5.

### 5.3.2 Interactor composition

A different way of implementing a new interactor is building one from available parts, by utilising the compositionality of the interactor abstraction. Suppose we need an interactor for selecting an element from the set of elements SET2 that is related through RELATION to a member of another set SET1. One possibility is to 1) let the user select an element from SET1, 2) generate the set of related elements i.e. SET2, and finally 3) to let the user select one of SET2's elements. If we already have an interactor for element selection, we can compose the desired interactor from two of these, as shown in Figure 71. The SELECTION1 interactor implements step 1, the RELATION connection between the input/send and output/receive gates implements step 2, and the SELECTION2 interactor step 3. At this point there are still some design choices left, both at the abstract and concrete level. For instance, at what point SELECTION1 is deactivated and SELECTION2 is activated, and which concrete interaction objects are used for each selection interactor.

Activation and deactivation of SELECTION1 and SELECTION2 is controlled by the way they (or their (sub)control states) are composed, as shown in Figure 72. Each interactor is con-

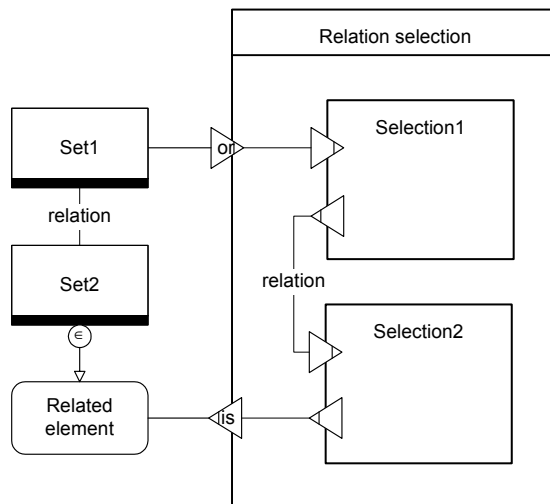


Figure 71. Selection of related element

sidered a state in a hierarchical Statechart, where each level in the interactor hierarchy corresponds to at least one level in the Statechart. Or rather, each interactor can be considered a multi-level Statechart. Activating the interactor corresponds to entering it as a state, and visa versa. In the active state, the gates of the interactor are ready to receive and emit values, and the internal control structure executes. The left model in the figure uses *and*-composition, giving parallel/simultaneous activation of the two selection interactors, while the right model uses *or*-composition, giving alternative/sequential activation. In the former case, both sub-interactors are always active as long as their super-interactor is, while in the latter case SELECTION1 is deactivated and SELECTION2 is activated when the transition's condition is satisfied, in this case when SELECTION1's input/send gate is fired.<sup>1</sup> In both models, the initial sub-state in an or-decomposed super-state is defined by the standard *default transition* mechanism in Statecharts, which is used when no direct transition exists. In the left model each default transition is redundant since there are only one sub-state in each compartment, while in the right model, SELECTION1 is preferred over SELECTION2 as the default start state.

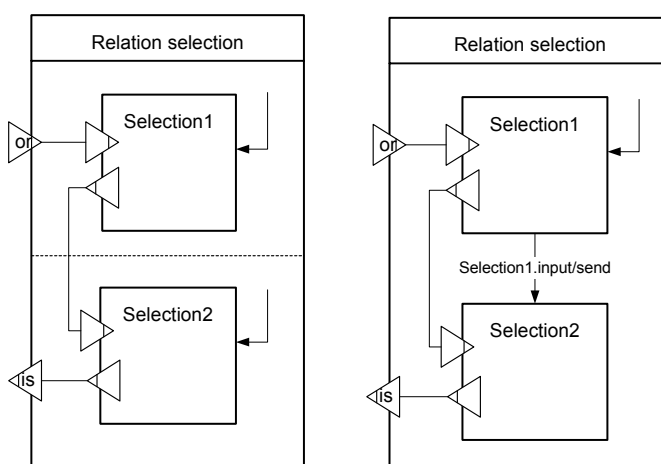


Figure 72. And- (left) and or-composition (right)

1. In most cases, the lines separating the substates in an and-decomposition can be deduced, since there anyway cannot be transitions across the lines. Hence, we adopt the rule that each connected subgraph of states are implicitly separated by such lines.

It might seem that and- and or-composition gives limited freedom to control activation, but since interactors and ordinary states can be freely mixed, any control structure can be realised. For instance, in the and-composition case, the activation of SELECTION2 can be delayed by inserting a new initial state and a transition with an appropriate condition, to give a behaviour somewhere in-between parallel and alternative activation. As shown left in Figure 72, the SELECTION2 interactor will be activated once SELECTION1's input/send gate emits a value, while keeping SELECTION1 activated. In the right model, the delay is accomplished by wrapping an interactor with another interactor with the same gate interface. The activation of the sub-interactor is controlled by the two transitions from the delay state. By coupling the events of the two transitions to appropriate interaction objects and the actions to window activation, the effect of popping up the interactor can be achieved.

In principle an interactor is independent of its implementation, i.e. it is a specification of a certain behaviour, which the implementation in terms of concrete interaction objects must adhere to. For instance, the composed interactors in Figure 72 may be realised in terms of frames or panes. If and-composition is used, separate panes may provide the appropriate behaviour, while or-composition can be implemented using either popup-windows or a single pane which is (alternatively) shared. In practice, the degree to which the concrete design affects the abstract interactor structure will vary, often a certain interaction style will warrant a certain interactor structure. For instance, the delayed activation shown above, is typical for menu based dialogues, where menu items are available for popping up dialogues. Sometimes the interaction style may even affect the task model for which the interactor structure is the solution. Hence, may be relevant to go backwards, i.e. derive the interactor specification of a concrete interaction object.

## 5.4 Modelling system functionality

Functions are mostly used for mapping between values, along connections or in-between the base and tip of gates. It is however possible to include standalone functions inside an interactor, either without arguments or with constant input. This is useful for providing access to global attributes or application services, i.e. information coming from the runtime context of the interface on the target platform, and not from the user.

Figure 73 shows two typical usages: The top standalone DISKDRIVES function provides the set of disk volumes, e.g. as input to a file browser. In the bottom left, a set of functions G1...GN are disguised as output/send gates of the GLOBALS interactor. These gates mediate a set of global attributes and the interactor is only used for grouping them in the model. Such an interactor mirrors the use of input devices, which provide values out of nothing, by providing the user interface with values from "nowhere", that may be emitted by an input/send gate.

Application or object services may be modelled in a similar way, as shown in Figure 74. An interactor provides a set of methods by means of pairs of gates, the input/receive gate in the pair accepts a set of arguments and the output/send gate returns the result. We may compare this interactor to an object, where each pair of gates corresponds to a message (port), or it may just be viewed as a set of global functions grouped together for clarity. By having separate argument and result gates, we suggest that the computation may be asynchronous.

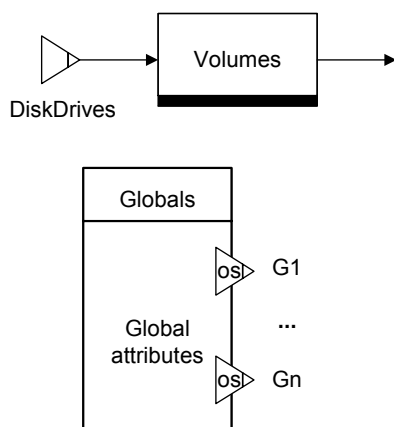


Figure 73. Standalone functions

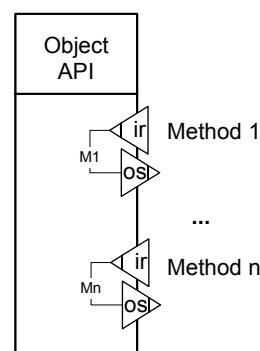


Figure 74. COM-like object

## 5.5 Interactor resources and parameters

An interactor may be seen as a specification of a design problem to be solved by composing existing interactors or by assembling concrete interaction objects from a toolkit. It may also be seen as a specification of an already solved problem, using the mentioned techniques, in the hope that it may match a future design problem. In both cases, the interactor may require support from the runtime context it will operate within.

The checkbox model shown in Figure 70, can be seen as one possible realisation of a boolean interactor. It is however not a complete solution, as there are two important parts missing: 1) an input device for driving the CONTROL states and 2) an output device for visualising the CHECKMARK and ARMING states. These parts must be provided by the context of the interactor, before the interactor will be completely executable. However, the checkbox model does not explicitly state that such devices are needed. In our interactor model, these should be defined and provided as *resources* or *parameters* of the interactor. Interactor resources are similar to the task resources described in Chapter 4, in that they define requirements for parts, that may be provided by the context. In the case of the checkbox, we would define two resources, e.g. called MOUSE and WINDOW as shown in Figure 75, detailing capabilities, like state space and sequences, that are used or needed by the checkbox implementation. The two resources are state resources, hence the state symbol is used.

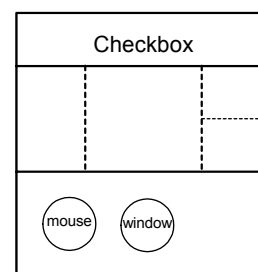


Figure 75. Interactor state resources

Interactor resources are typically used in two ways. First, they provide a link to the concrete interaction perspective by stating the specific *requirements* that must be fulfilled by its implementing toolkit interaction objects. As requirements, they *limit* the context in which a particular interactor may be used, e.g. only within a particular window-based runtime environment or on a pen-based system. This usage is suitable for cases where the interactor represents a particular native implementation, and the requirements should be stated in terms of native concepts, e.g. Java classes or COM interfaces. Alternatively, resources can be used to

define parts of the interactor implementation that must be provided and can be replaced before using it, e.g. supporting functions or sub-interactors. This makes the interactor implementation more *widely* useful, since its behaviour to a certain extent can be adapted to a new context. In both these cases we need some way of defining the requirements of the resource and how a bound resource fulfills those requirements.

In the checkbox case, the mouse and visual resources could be used in either way, i.e. either *limiting* the use to a specific mouse device or *widening* its use by letting the context provide the input device as a parameter. As the first kind, we would require that there be a specific mouse and window pair providing the required set of events for the CONTROL states and a window pane capable of visualising the CHECKMARK and ARMING states. For instance, in a Java implementation we could require a `java.awt.Panel` for the window and use a `java.awt.event.MouseListener` for driving the mouse part. In addition, a pair of bitmaps could be required for the checkmark and two background colours for the arming. These objects would provide a link to the underlying Java runtime platform. If the mouse and window resources instead were used to define the formal type or characteristics of the parameters, i.e. the second usage, we would need to gather the requirements assumed by the model, or state them explicitly.

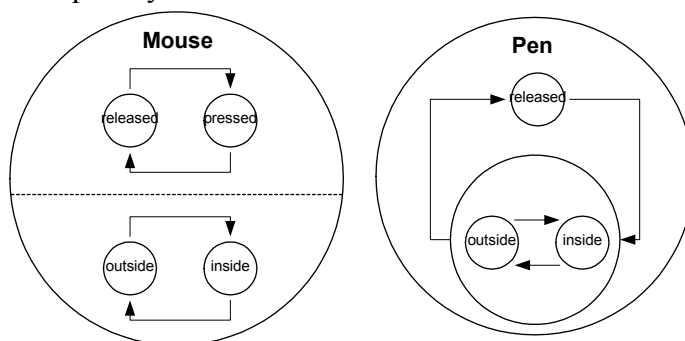


Figure 76. Mouse and pen models

From the checkbox model in Figure 70, it can be seen that the mouse resource should (at least) provide a state space including states for the PRESS/RELEASE and OUTSIDE/INSIDE condition pairs in the model, and the visual resource should distinguish between two pairs of ON/OFF states. If analysed more closely, we can see that the checkbox model assumes that a certain (temporal) sequence of the states or conditions is possible, e.g. that PRESS & INSIDE can be satisfied together and that RELEASE may follow PRESS. In this particular case, both a mouse and pen can satisfy the requirements, e.g. with models as shown in Figure 76. Different behavioural requirements might however, make the mouse the only candidate. For instance, if the checkbox needed to distinguish between INSIDE and OUTSIDE for both PRESS and RELEASE to provide appropriate feedback, only the mouse could be used, since the pen's INSIDE state is only defined in the PRESSED state, according to this model.

To be able to ensure that a provided resource is compatible with the requirements, we would first have to completely spell out the requirements and then prove that the provided resource is a complete and consistent implementation. In the case of the checkbox and the mouse, we would first have to derive a state machine that could “drive” the control part of the checkbox through all its states. This machine would represent our resource requirements and would actually be close to the pen interactor shown in Figure 76. To make sure it was safe to use the mouse interactor instead, we would have to show the correspondence between states and transitions in the respective interactors. In the general case, it is very hard if not

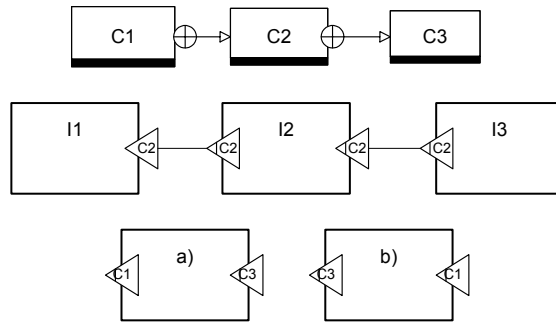
impossible to prove that one state machine can replace another.<sup>1</sup> As a complicating factor, we cannot expect every relevant resource to be naturally described in terms of state machine concepts, so even if a general proof procedure exists, it would not completely solve the problem of defining and replacing parts of a state machine. Instead, we would have to rely on the developer to provide a mapping or interface to glue a replacement resource into place.

The checkbox model is rather simple, since it only involves states, conditions and transitions: no values, gates, connections and dataflow. To make it practical to give a similar treatment of interactor resources, with formally defined requirements and proofs of substitutability, we limit the *essential characteristics* of interactors to the set of gates, their types (input/receive, output/send, input/send and output/receive) and the types of the resources (interactors, functions and domain sets and elements). These are the static part of the interactor definition, leaving out sequencing. Limiting the essential characteristics in this way has three implications: First, a composite interactor should only rely on the gates of sub-interactors and the values they emit. Second, to be able to replace a resource, an interactor need only have the same gate and resource signature. This makes it practical to perform static consistency checks. Third, by reducing the analysis to static characteristics, we might unfortunately accept a parameter substitution that should be rejected.

This simplification of the essential characteristics of interactors still leaves some problems concerning substitutability, consistency and completeness. Consider Figure 77, where a simple concept hierarchy C1, C2, C3 and chain of interactors I1, I2, I3 is shown. All the gates in the chain of interactors are labelled with C2, to indicate that they have the C2 type. The question is which of candidates interactors a) and b) at the bottom of the figure, can replace I2? The b) interactor is consistent with the context of I2, since it may receive any value provided by I3 and will only send acceptable values to I1. However, the resulting model will not be complete, since some acceptable and expected values of I1 can never be sent to it. Candidate a) has the opposite problem: it receives a too limited set of values from I3 and sends a too wide range of values for I1. However, it will at least fulfill I1's needs and accepts all values received from I3. The case where an interactor is too generic for a given context, is actually quite typical, e.g. an integer device interactor may be used in a context where only integers in a given range are acceptable. The standard way of handling this is either to design and implement a new special-purpose integer range interactor or to provide a validation function that alerts the user if the value is outside the acceptable range.

An example of a useful generic interactor is shown in Figure 78. This is a parameterized version of the integer interactor in Figure 58, where the two functions have been defined as replaceable resources or parameters. The generic interactor is a device interactor for any type, given appropriate replacement functions. Although not explicitly shown in the model, the unparsing and parsing functions must accept and emit the same value type, and the two input and output gates must accordingly receive and send this same type.<sup>2</sup> This is truly a

- 
1. As long as there is finite state space and a finite set of loops, it is possible to use exhaustive search to perform the proof. However, this is not practical for larger state machines.
  2. This means that an explicit formal interactor definition must rely on a type variable to relate the gate and function signatures, e.g. something like `interactor(output/receive:<type>; input/send: <type>; resources: parser(string)=><type>, unparsed(<type>)=>string).`

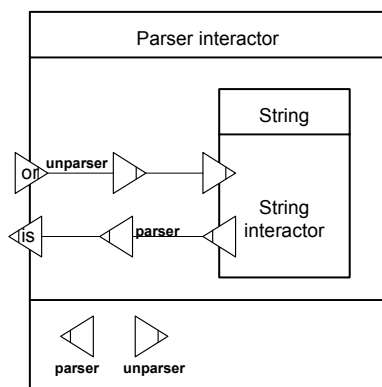


**Figure 77.** Problem of substitution:  
Which of interactors a) and b) may replace I2?

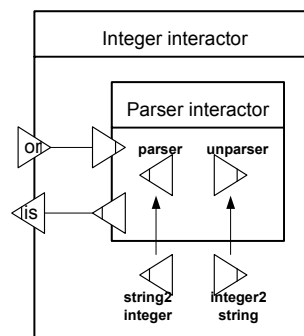
very generic interactor, and has been used to handle diverse types like date & time, colour, email addresses and contact names, by appropriate instantiation.

Interactor resources are placeholders for values, much in the same way that gates hold values. The interactor defining a resource may provide a (default) value for it, which may be replaced by the context in which it will be used. Hence, the resource provides support for *customising* the interactor’s functionality, since the model is complete but modifiable. Alternatively, the interactor may require that the context provides the value, before the interactor is used. In this case, the interactor and resources are used to provide a *template* mechanism, since without suitable parameter instances the model will not be complete.

The generic PARSER INTERACTOR interactor in Figure 78 is a template in this sense, and the simplest way of instantiating it is by providing replacement resources, in this case two functions, in a *wrapper* interactor. To implement the previously shown integer interactor, we have to provide integer parser and un parsing functions, as shown in Figure 79. We see that the composite interactor contains the generic interactor (signature), and provides two special-purpose functions as replacements, to achieve the desired functionality of integer input and output. The two functions are *constants* that are bound to the parser and unparser resources, within the context of the outer interactor. The PARSER INTERACTOR signature and the bindings collectively represent a new interactor instance, built from the (template) definition shown in Figure 78.



**Figure 78.** Generic string parsing and unparsing interactor

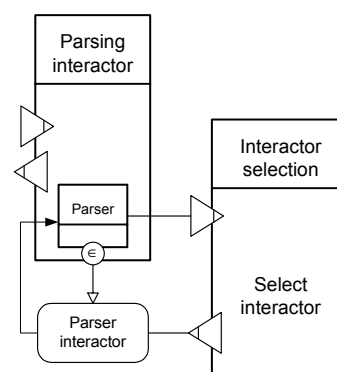


**Figure 79.** Interactor instantiation, through the use of resource binding



Interactor and gate resources are typically bound to actual values during model editing, like in the above case. The template can be seen as a convenience for the designer or developer, while not providing the end-user with any functionality or power. However, just as customisation of a product may happen during build, deployment or while in use, resources may be provided through model editing, interface configuration, or dynamically during use. To support the latter kind of customisation, the designer must provide the end-user with special interaction objects that operate on the dialogue/interactor domain instead of the application domain.

An example would be an interactor used for selecting the format or language to use for inputting integers, e.g. a drop-down listbox. The selected value would be bound to the resource of the integer interactor, hence changing the accepted syntax of integers. An interactor model for such a pair of interactors is shown in Figure 80. The PARSING INTERACTOR in the figure is assumed to be a parameterized version of the model in Figure 55, now with a PARSER (interactor) resource responsible for converting to and from the string representation. The PARSER resource has a dual use in this particular model. First, it serves as the definition of a type of interactor with a specific signature, or in RML terms, an interactor concept. The extension of this concept is used as input to the selection interactor in the figure. Second, it is a named resource, which is bound to the value of the input/send gate of the selection interactor. The effect is that the user may use the INTERACTOR SELECTION interactor to select the parser used by PARSING INTERACTOR. Many kinds of customisation can be modelled in this way: A generic interactor with a resource parameter, which is bound to the input/send value of the interactor used for customisation.



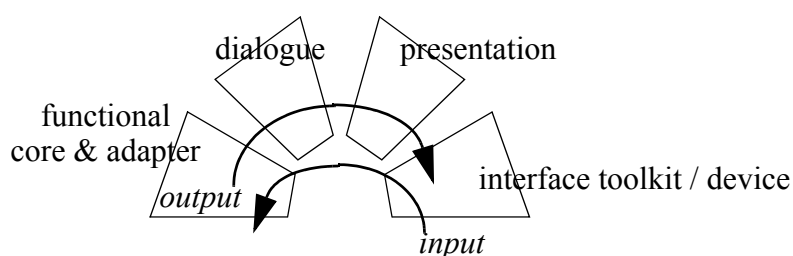
**Figure 80.** Interactor customisation

## 5.6 Acts and functions

The goals that the user wants to achieve and the tasks he believes are necessary is represented in the task model. The dialogue model on the other hand, describes in detail how a design solution allows the user and system to cooperate in reaching these goals by performing the tasks. Many tasks are better left to the system/functional core, and some will necessarily have to be performed by it, since some parts of the real world is not directly within the reach of the users. The users' main role with respect to these tasks will be to indicate that the system should perform them. For instance, to send an email, it is clear that the system has to set up the network connection and transfer the message for the user, since the user cannot physically perform these tasks. The user on the other hand should indicate the recipient, author the message, and tell the system when to send or queue it. The system's work will be partly performed by interactors, as mediators of information between the user and system in both directions, and partly by functions, which operate on domain objects and the underlying infrastructure. Functions were introduced in Section 5.2 and will be detailed in Section 5.6.4, while we at this point will discuss the act of activating functions.

### 5.6.1 Acts

Interaction can be viewed as a dialogue between a conscious user and a system, driven by the dialogue component's interpretation of the user's manipulation of the input devices. In the Arch model an interactive system is divided into four conceptual parts, as shown in Figure 81. The *functional core* and its *adapter* contains the main domain specific functionality, while the *dialogue* and *presentation* parts implement the abstract and concrete dialogue structure and elements. Finally, the *interface toolkit* provides the platform's devices for input and output. According to the Arch model, input events from an input device are interpreted by the dialogue component and may result in the activation of a function in the core. This is illustrated by the left-pointing lower arrow in Figure 81. The result may ripple back through the dialogue and presentation components, before being sent to the output device, as illustrated by the right-pointing upper arrow.



**Figure 81.** The four components of the Arch model

Different interaction styles make this dialogue view more or less explicit, e.g. a command line interface is based on a repeated two-step dialogue of typing in calls to the functional core and displaying the result. The activation of a function is in this case explicitly represented by the command line string. A direct manipulation interface, on the other hand, lets the user visually manipulate domain objects, and makes the dialogue implicit by replacing the linguistic utterance with mouse manipulation. It is nevertheless useful to explicitly represent what the act of typing or mousing means, since functionality like undo and redo will need to make function activation explicit.

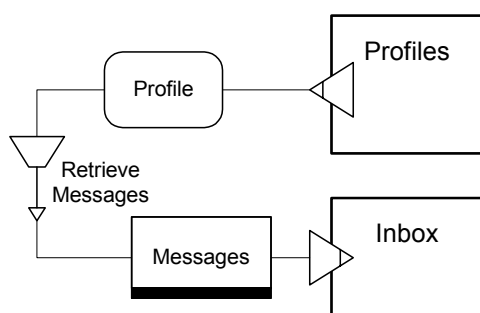
We define an *act* to be an object that represents the user's desire to have the system perform a function in order to reach some goal, or in more concrete terms, to apply a function to a set of arguments. An act will usually correspond to the goal of a user task, even if this goal has not been explicitly modelled, either because the task model is of high granularity or because the goals themselves are not explicitly modelled. Acts are used when we need an explicit representation of a function and a set of arguments as an element, to capture the systems interpretation of interaction. Many functions may be applied without such an explicit representation, either because they are not very domain specific and need not be transmitted all the way to the functional core, or because they have little relevance for the user. This is typical of standard services like directory queries and string parsing, which are more or less taken for granted. Conceptually they may still be part of the functional core, but they are not "worthy" of much attention from the user or the dialogue modeller. In other cases, function invocation is more apparent for the user and more complex, and there is need for introducing the concept of an act:

- The user interacts with a visual representation of an act or command.

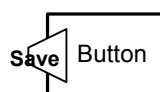
- The function's arguments may not be fully provided at once, and may need to be completed by either using contextual information or prompting the user.
- The function should be executed in a different context, which is detailed in a different part of the model.
- The function should not be applied to its arguments, but instead handled as a piece of executable data, e.g. if macro recording or scripting is turned on.

We have chosen to use the function symbol without a tip for indicating an act, as the base may be thought of as representing (the presence of) the arguments while the removed tip suggests that there is no result (yet), whether slightly or indefinitely delayed. The corresponding function definition will be separately specified, as discussed in Section 5.6.4. The tipless function symbol may represent a possibly incomplete function application and is similar to a *curried* function, i.e. a function where some of the arguments have been provided, and some remain to be. Technically, an act consists of a function name, and a set of named and typed arguments, some of may already be bound to a value. The actual performance of the function must be delayed until all arguments have been collected. For the purpose of handling the return value, an act in addition contains a reference to its origin/context.

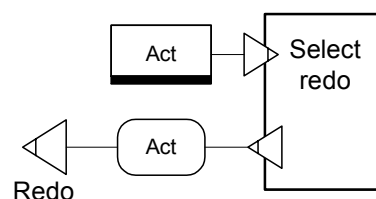
An act is used in diagrams in two ways. First, it may replace a function, when we want to make the act of activating a function more explicit, e.g. when the user is conscious of the function application. In this case of act invocation, the act is connected to a standalone tip, to create the impression of a base and tip torn apart, or an exploded function, as shown in Figure 82. The arguments are still attached to the base, while the output is attached to the (dislocated) tip. The main difference is that a function is activated immediately, while the act represents a potential function activation. When the act is triggered, e.g. by the availability of input, the function (name), arguments and origin are combined into an act (instance). This instance is attached to an asynchronous Statechart event or action, that may be received by a corresponding function specification/implementation in the originating inter-actor's context.



**Figure 82.** The Retrieve Messages act modelled as a torn-apart function



**Figure 83.** Act input device



**Figure 84.** Act as set and element

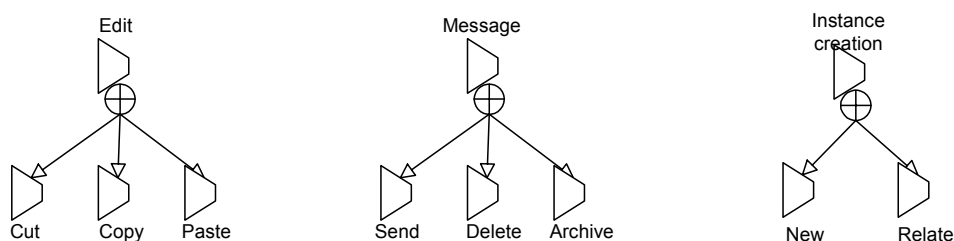
When the result of invoking an act is not used, the tip will not appear at all, and the act symbol will seem like a dead end. The result of the act may however be used in the context where the act event is received and invoked. Figure 83 shows a model of a button used for invoking a save function by emitting a Save act. Presumably the act will be received in a context where the data or document to save is available.

The second way of using the act, is as an ordinary data element, e.g. when selecting a previously performed act to undo or redo it, as shown in Figure 84. In this case, the ordinary set and element symbols are used, to avoid confusion. To execute such an act, it must be fed into a second order function, i.e. a function taking another function (or act) as input. In the figure, this is an ordinary function, but the act symbol could have been used, e.g. to indicate that redo is itself a conscious action, which later may be redone (or undone).

## 5.6.2 Act classification

These two alternative mechanisms allow either executing a specific named function, by emitting and receiving an act as an event, or any function by giving a higher-order function an act as argument. A combination of these is also possible, by making a higher-order function selectively receive act events within a certain class. We treat acts as ordinary RML concepts, which may be classified and generalised/specialised using the standard RML constructs. Several act classifications may be relevant, as shown in Figure 85.

- We may classify acts based on their role in standard tasks, e.g. copy, cut and paste acts support the task of editing documents and hence can be considered *special* edit tasks. This classification may be the basis for grouping acts in menu bars.
- The arguments may be used for classification, e.g. every act taking a message as the ‘message’/first argument, or the whole argument list may be used for defining a classification hierarchy.<sup>1</sup> This classification may be the basis for contextual menus, which are activated by clicking on the main argument of an act.
- The result or the goal achieved may be used for classification, e.g. creation acts for all acts creating new objects, from initial values. This classification is relevant for understanding how tasks may be accomplished.



**Figure 85.** Act specialisation, based on task, argument type and result

The possibility of classifying acts makes it possible to augment similar acts with appropriate functionality. For instance, an email client may default a missing mailbox argument of an act to the one currently selected in a contextual mailbox interactor. Another possibility is defining interface frameworks or templates for invoking standard classes of acts, within a certain interaction style. Classification of acts is similar to classification of tasks as discussed in Section 4.7.7, and provides a means for building generic models that may be instantiated with domain dependent elements.

1. This is similar to how generic functions are partially ordered in the method invocation mechanism in Common Lisp.

### 5.6.3 Act invocation

The functional core, which conceptually resides outside the user interface, is usually responsible for performing the acts, or rather, applying the function to the parameters specified by the act. In a flow-oriented model, the functional core can only be reached by propagating values along connections out of the layers of interactors, i.e. up through the top of the interactor hierarchy. In an event-oriented model, the functional core can alternatively be reached by sending events, which are received outside the user interface. We will adopt a hybrid model for invoking functions: acts are attached to events that are broadcasted, the events may be received and re-broadcasted by functions in the containing layers of interactors and finally may be received by the functional core. Hence, the *effective* implementing function is a combination of the layers of functionality provided by the layers of interactors. In the simple case, the function corresponding to the act will be applied to the arguments directly. Alternatively, the act may be transformed, arguments may be added or replaced, and the act may be re-broadcasted attached to a new event, to be received by a different function. An alternative way of viewing this is to think of the functional core as consisting of layers which add, replace or modify (existing) functionality, some of which are added by the user interface, as parts of interactors.<sup>1</sup>

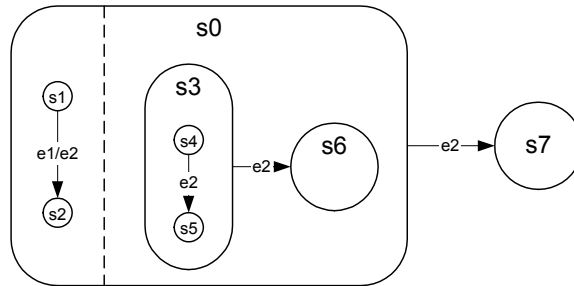
Since each act may be handled by several function, the effective functionality corresponding to an act is a combination of several parts of the model. The order of execution of these parts, is most likely important for the result, hence the broadcast mechanism must be predictable. In classical Statechart, events are broadcast throughout the whole state machine, and may trigger any transition. Outer states have *priority*, presumably because sub-states should not be allowed to change the higher-level behaviour. Giving outer states priority ensures that adding a sub-state does not prevent transitions in its super-state from being triggered by an event. Hence, invariants that are specified at one level, are preserved when the model is decomposed into an additional level. In UML the rule is the opposite: lower-level transitions may “override” higher-level ones, presumably to allow a sub-state to change inherited behaviour.<sup>2</sup> This may however, invalidate invariants from the level above, and makes it difficult to analyse the state machine. We have adopted a mixed rule: an event is first broadcast within the immediately containing super-state of the state emitting the event, then to the super-super-state at the higher level, etc., until it triggers some transition. At each level the top states have priority, so a transition in the super-state of the originating state will have priority above a transition in a sibling state. With this rule it is only possible for a sub-state to grab locally generated events i.e. those originating within that sub-state. Hence, it cannot change the behaviour specified at the level of its super-state.

The difference between these three priority rules is illustrated in Figure 86. The E1 event will trigger the transition from s1 to s2, which will generate the E2 event. With the priority rule of classical Statechart, this will trigger the transition from s0 to s7, since this is outermost transition labelled by the E2 event. UML’s priority rule will trigger the transition between s4 and s5, since this is the innermost transition labelled by this event. With the rule we have adopted, the transition between s3 and s6 will be triggered, since this is the outermost transition inside the first super-state containing a E2-triggered transition. As men-

---

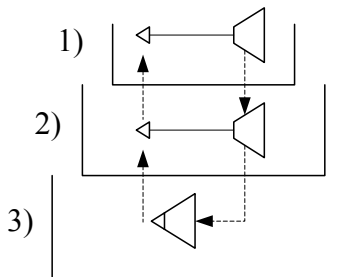
1. The mechanism is inspired by the inner mechanism in Simula and the method combination feature in the Common Lisp Object System, the full meaning of a function is defined by a combination of individual method bodies.

2. Section 2.12.4.7 of UML Semantics



**Figure 86.** The effect of different transition triggering priorities

tioned, our choice of rule is something in-between the classical Statechart and UML. We want to limit the interaction between distant states, while preserving the possibility to reason about invariants in the machinery.



**Figure 87.** Cascading acts

The act symbol looks like a torn-apart function, and we want the behaviour of act invocation to be as similar to function application as possible, in the way argument values are received at the base and the resulting computed value is sent out of the tip. Hence, although the computation may be performed in the functional core, the flow of values should hide this. Figure 87 illustrates the flow of acts and return values. The top (1) two-part act symbol generates the first act event, which is received by a second (2) two-part act symbol in the context. The act event is re-broadcasted and this time it is received by an ordinary function (3), which finally performs (the core of) the desired function. The stippled arrows indicate the act event flow. Since the tip and base pair, representing act invocation, can replace a simple function application, it must eventually emit a return value. This means that the receiver of an act event, must feed its return value back to the originating act invocation, as shown in the figure. The flow of return values will be the reverse of the flow of events. At each act invocation tip, the return value may be manipulated before being propagated further. Note that the result value will also be propagated along the connections of a tip, so in the case of the middle act invocation in Figure 87, the result will both travel out of the tip to the left and up to the initial act invocation.

Although this adds some complexity to the model, there are advantages of letting several function implementations combine to give the full meaning of an act:

1. The whole hierarchical context of the act may determine its interpretation.
2. Contextual information may be added to the act at each level.
3. The core function may be adapted or wrapped to provide more flexibility.

Consider the case where a non-graphical functional core is to be extended with a graphical user interface. Many core functions will have to be augmented, either by adapter functions or by a complete functional layer. The new graphics-aware functions will perform its part, pass the relevant arguments on the existing functions and adapt the result to the graphical context. For instance, a Make New Folder act will include an icon position in the graphics layer, but only pass on the name to the existing functional core. The resulting folder object may be annotated with the position, before being presented to the user.

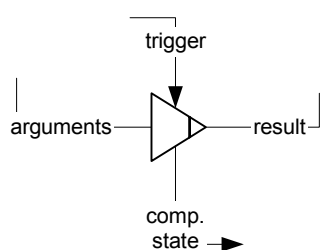
### 5.6.4 Function and gate decomposition

Functions may for most purposes be considered atomic actions, with no structure. In some cases however, the user may perceive the function execution more as a process than as a simple computation, and the designer may want to detail the function's behavior. There are several reasons for making the computational process transparent, for instance:

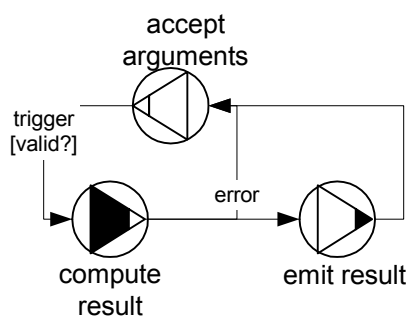
- The function may find that some arguments are invalid and notify the user,
- it may reveal its computation state by updating the user interface and
- it may interact with the user, to let him control the functions execution, normally to provide a means of cancelling or skipping it.

All these considerations suggest that functions are atomic for some purposes, but for other purposes should be considered to be complex, not unlike interactors. Accordingly, we must provide constructs for describing the aspects of functions that are necessary to expose to the user.

The objective of a function is to compute a typed result from a set of typed arguments. In our dialogue model, the argument values are provided through connections attached at the function's base, while the result is received by another connection through the tip. For simple functions there is little reason to monitor the computational process, but for more complex functions this may be important for the user. To monitor and control the function's computational process, connections may additionally be attached to the side of the function. The crucial question is whether the user should be aware of the state of the computational process, e.g. through a progress dialogue. If the answer is yes, the function's computational process may be modelled as a state machine as shown in Figure 89.



**Figure 88.** Function usage: input arguments, trigger, monitoring and result output

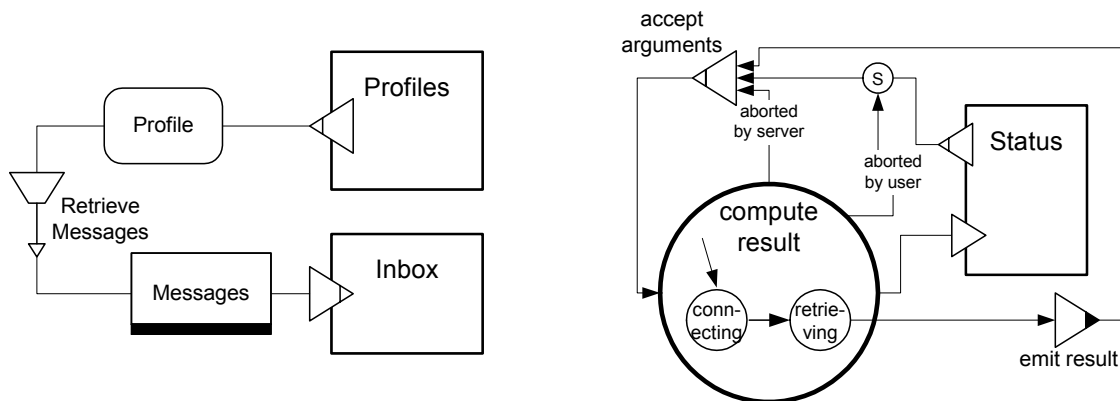


**Figure 89.** Function decomposed into three main states

The idea is that the function cycles through three main states, each of which is related to one connection in Figure 88. The three states have been given special function icons, to suggest which part of the computation they are concerned with. Throughout this cycle, the current state may be monitored through the fourth connection labelled "comp.state" in Figure 88.

1. ACCEPT ARGUMENTS: Values are collected through the connections at the function's base, i.e. the connection labelled "arguments" in Figure 88. The empty base and tip of the function icon suggests that it is in the beginning of the computational process. Arguments may be "eagerly" validated as they are received and the user notified if something is wrong.
2. COMPUTE RESULT: After a trigger event is received, through the connection labelled "trigger" in Figure 88, and the all the arguments are validated, the computational process begins. The filled base and empty tip of the function icon suggests that the arguments are present. This process either results in an error or a value in the tip. The trigger corresponds to the value entering through the connection attached to the function's side, at the top left in Figure 88.
3. EMIT RESULT: The arguments have been consumed and the tip has been filled, as indicated by the function icon. The value is emitted to the receiving connections attached to the function's tip, i.e. the one labelled "result" in Figure 88. The function returns to the state of accepting arguments.

Figure 90 shows how a complex function, RETRIEVE MESSAGES, can be used and decomposed. In the left fragment, the function receives a PROFILE element which is input by the user and returns a set of MESSAGES which is output by the INBOX interactor. The decomposed function at the right contains the same three main states as those shown in Figure 89, and in addition decomposes the COMPUTE RESULT step into two sub-states. The computation either results in a value in the tip, is aborted by the server or aborted by the user. During the execution, the user is presented with the status of the main states, modelled by connecting the COMPUTE RESULT state to the output/receive gate of an interactor. I.e. the computational state the function is in, is presented to the user. In addition, the user may select the passive ACCEPT ARGUMENTS state to abort the operation. This is modelled using the Statechart select transition mechanism, using a circle with an "S" inside. This mechanism supports explicitly selecting the state to enter, rather than using some other triggering event.



**Figure 90.** Use (left and decomposition (right) of Retrieve Messages function

All functions must contain the three main states, and these may include substates detailing their behaviour. Interactors may be included to present the state of execution to the user and provide means of controlling it. Although not shown, domain fragments including concepts, elements, relations and functions may be included to specify how the tip's value is computed, based on the input. Note that the goal of decomposing functions is not to specify algorithmic details of how values are computed, but rather describe the parts of the process



that must be exposed to the user. Hence, the ability to decompose functions should not be used to model arbitrary computations.<sup>1</sup>

As described in Section 5.2.1, gates can be considered functions playing specific roles with respect to an interactor. Although gates default to the identity function, they can perform computations, be decomposed and refined if that is necessary for modelling an interactor's behaviour. Of particular interest is how gates receive, retain and emit values, corresponding to when the output and input behaviour of the hosting interactor are active. Recall the four main roles a gate can play, output/receive, input/send, input/receive and output/send, where the first two are most important as they are more directly experienced by the user:

- Output/receive gate: The base value represents the requirement that the interactor should make this value perceivable for the user. The tip represents the actual value the user perceives.
- Input/send gate: The base value represents the user's input value, while the tip makes this input value available for the system.

Most window-based interaction objects constantly show both an output and an input value, i.e. both the output/receive and input/send gates will have a value, which will often be the same. For instance, a checkbox will always show either a true or false value; the initial value is the system's output and subsequent values the user's input. In detail, the steps should be as follows:

- The system's output value is propagated to the base of the output/receive gate
- The interactor updates the checkbox state and sets the output/receive gate's tip accordingly. The output/receive base is cleared to indicate that the request for output has been fulfilled.
- Since the output value at this point is also the input value, the base of the input/send gate is also set.
- The first time the user clicks the checkbox, the output/receive gate's tip is cleared, since the checkbox' value now is considered an input value.
- Each time the user clicks the checkbox, the base of the input/send gate is set accordingly and the value propagated to the tip, so connected interactors can react to the value change.

Most graphical device interactors should behave like this, while other behaviours or media may behave differently. For instance, the press of a button is a one-time event, while standard interaction objects/widgets retain their value until explicitly modified by the user or system. In the case of speech synthesis, the output value does not persistent as it does for display based output, so the output/receive tip will quickly lose its value. To describe the various possible behaviours we need to be more explicit about what happens when values are propagated in terms of the state machine inside gates, using the technique described above.

---

1. Other notations like Petri Nets or flow diagrams may be more suitable for a full specification.

Figure 91 shows two gates, the left an output/send gate and the other an output/receive gate, in a four-step propagation sequence, as follows:



**Figure 91.** Propagation of along connection

1. The base of the left interactor's output/send gate holds a value, which is internal to the hosting interactor.
2. This value is made available in the tip outside the interactor and propagated to the base of the connected output/receive gate of the other interactor.
3. The tip of the right interactor's output/receive gate has received the value, indicating that the user can perceive the value.
4. The right tip's value is removed, if the output value is ephemeral.

The two gates and the connection in-between them cooperate in achieving this particular behaviour. In step 1, the base of the output/send gate either receives a value from an internal connection of the left interactor, or is directly controlled by the interactor. In step 2, this gate emits the base' value through its tip, and the connection propagates the value to the right gate's base. The connection may of course include a function which may or may not compute a new value. In step 3, notice that although the internal value at the base is still in place, the tip is not refilled. In the typical case this indicates that the value has not changed, hence, there is no need to propagate it. Several other behaviours exist, e.g. re-propagate the value at once, wait a certain time, or wait for a specific event internal to the hosting interactor. We also see that in step 4), the value in the right gate's tip is removed, indicating that output medium is not persistent. In this case, the only other option is to retain it, which would require the right interactor to let the user continuously perceive the value. As mentioned, this is typical of standard widgets, while the first behaviour may be a better model of animation or sound.

Flow of values in the input direction is controlled in the same way. An input/send gate controls when a value should be propagated as input to the system, the base reflecting the value that an interactor represents as input, while the tip triggers the propagation of the input value. Input/send gates often provide a buffering behaviour, either by waiting for the value to change or a specific confirmation event, and validating the input using appropriate validation tests.

## 5.7 Conclusion

We have presented a language for modelling abstract dialogue, DiaMODL, which is designed as a hybrid of a Pisa interactor and Statecharts languages. As a hybrid, DiaMODL is a highly flexible notation with few basic concepts. DiaMODL is tightly integrated with the RML domain modelling language, which is also used for defining the DiaMODL constructs.

### 5.7.1 DiaMODL and the representation framework

DiaMODL is designed for modelling the abstract dialogue of the user interface. In terms of the classification framework introduced in Chapter 3, it covers the left part of the solution-oriented perspective, by focusing on the abstract structure of the software solution. DiaMODL is hierarchical and thus targets several levels of granularity, and we have given examples of using DiaMODL for modelling small and medium-sized parts of the user interface. DiaMODL includes constructs with varying formal strength, in the sense that more or less of the dialogue is precisely defined. For instance, it is possible to initially capture only structural aspects and later specify sequencing.

Below we will discuss how DiaMODL supports the 6 movements in the design representation space introduced in Section 3.6:

- Movements 1 and 2, along the perspective axis: From problem to solution and from solution to problem, respectively.
- Movements 3 and 4, along the granularity axis: From down the hierarchy from high-level descriptions to lower-level ones, and up the hierarchy from lower-level descriptions ones to higher -level ones, respectively.
- Movements 5 and 6, along the formality axis: From informal descriptions to more formal ones, and from formal descriptions to less formal ones, respectively.

For DiaMODL the relevant movements within the representation space are movements to and from task models and concrete interaction objects, up and down the hierarchy, and to and from informal design descriptions like sketches. We will now discuss these movements in turn, starting with the movements between models of task and dialogue, i.e. movements 1 and 2 between the problem- and abstract solution-oriented perspectives.

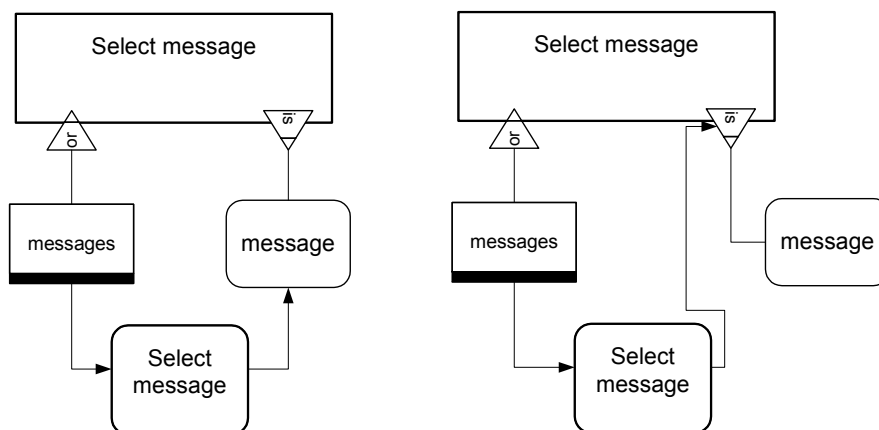
#### Movements 1 and 2

Task and dialogue models are similar in many respects, e.g. they both describe dynamic processes, how processes enable each other through control and information flow. The important difference is concerned with what processes they describe. In task models the processes represent the actions that the users consider are necessary for reaching their goals. The dialogue model, on the other hand, describe how software processes are driven by the user's interaction. In Chapter 3, "Design and representation" the task model was described as problem-oriented, while the dialogue model was described as solution-oriented. Hence, it is important to separate the two, to avoid using task modelling for dialogue design or visa versa. Nevertheless it is important to be able to relate task and dialogue models to each other.

First, during design the task model can be seen as representing a set of requirements for the dialogue, and the dialogue model would need to meet these. Hence, it should be possible to express how the dialogue supports a particular task, i.e. makes it possible to reach a certain goal. Second, with the difference in focus in mind, it is important to ensure that the task model does not implicitly assume a design that has not been decided upon. I.e. the task model should not be used to describe action sequences that are forced upon the user by a certain design, without having made that design explicit. We therefore need to be able to

refer to the dialogue that a task assumes. Using TaskMODL and DiaMODL, the relation between task and dialogue may be expressed in several ways:

- The interactor that a task assumes may be included among the task's resources.
- The pre-conditions for performing a task may be related to triggers of interaction.
- The post-conditions of a task and the result of an interactor may be related.



**Figure 92.** Combining task and interactor models

Figure 92 shows two ways of expressing how an interactor supports a task, by combining elements from each language. The upper part of both the left and right fragments correspond to Figure 62, and expresses that the SELECT MESSAGE interactor support selection of an element in a set. Both fragments express that a set of MESSAGES is used as an enabler for both the task and interactor. The left fragment in addition expresses that the interactor is used to select the very MESSAGE element that is the goal of the task. The right fragment instead expresses that the goal of the task is to trigger the input of the message element, by means of the interactor. The difference is subtle: describing how the interactor supports the task vs. the how the task of operating the interactor is structured. The former is relevant when validating that a design fulfills the functional requirements, while the latter is relevant when evaluating the usability of a design, including the efficiency.

For designing a functionally complete dialogue, it is important that the task model may be actively used. A DiaMODL model can utilise most of the information provided in a TaskMODL model. The static part expressed using RML, may be directly used and further refined. Contextual information gives rise to interactor output devices, tasks that are performed for each element in a set may be supported by a selection interactor and a task specific interactor, etc. Task sequencing can be implicit in interactor connections or translated to explicit state sequences using transitions. Task goals may be reached by defining appropriate functions that receive their input from supporting interactors. Although DiaMODL has no concept of specialisation, task classification may be utilised for designing generic dialogues with structure and behaviour that are tailorable by means of interactor resources.

Moving from dialogue to task structure, is also relevant, particularly when evaluating a design. As mentioned in Section 4.8.1, it should be possible to express in TaskMODL the possible task sequences for a particular dialogue. The Statechart-based approach provides parallel and disjoint composition that fits fairly well with the aggregation and sequencing

constructs used in TaskMODL. As shown right in Figure 92, a TaskMODL fragment may be linked to a DiaMODL fragment, to indicate how a task includes actions for triggering dialogue elements.

Movements 1 and 2 may also be performed to or from models of concrete interaction, respectively. The examples in Section 5.2.2 and Section 5.2.3 shows that most concrete interaction objects may be modelled as abstract interactors. Section 5.3 and Section 5.5 shows that it is possible to add detail if necessary, to capture special features of concrete interaction objects. This means DiaMODL provides support for movement 2, i.e. moving from concrete interaction to abstract dialogue. The topic of formalising concrete interaction objects will be further discussed in Chapter 6, “Concrete interaction”.

As indicated in Section 5.2.2 and Section 5.2.3 most atomic interactors have a corresponding *prototypical* concrete interaction object, i.e. a concrete interaction object that is a natural candidate and typical choice when moving to concrete interaction. A GUI builder may for instance visualise an abstract interactor as its prototypical concrete counterpart, if the developer has made no explicit choice. In addition, previous sections indicated many of the alternatives that exist, when moving from abstract to concrete interaction.

### Movements 3 and 4

DiaMODL provides good support for hierarchical modelling., i.e. movements 3 and 4. The gate interface at one level can be seen as the specification for the level below, and provides some guidance for constructing the decomposition. Both the interactor concept and Statecharts support composition well. Given a set of interactors, it is easy to compute the composition and its minimal interface and later add gates and interactor resources, to provide access to the internal structure. The flexibility of Statecharts to include transitions that span subtrees makes reasoning about behaviour complicated, as triggering a transition may require exiting and entering large hierarchical structures both above and below the transition.

### Movements 5 and 6

Movements 5 and 6 are concerned with the interaction between formal and informal design representations. DiaMODL is not targeted at capturing common sense understanding of how user interfaces behave, and hence cannot be said to provide particular support for going from informal representation of abstract design. DiaMODL is however based on few, simple and generic constructs, so there should be little possibility for conceptual mismatch which would require complex translation from informal to formal concepts. With its abstract nature, we expect most movements from the formal interactor-based dialogue models towards informal representations, to go through more concrete representation. This may include generating concrete interface examples to highlight abstract features of the design, such as containment structure. To indicate choice and sequencing it may require animating the concrete example, to make the sequencing expressed at an abstract level understandable.

The support of the four first movements is in general fairly good, while movements 5 and 6 are more difficult to support well, due to DiaMODL’s abstract nature. In particular, we have shown how the correspondence with concrete interaction objects supports moving back and

forth between abstract and concrete design elements. This will be further explored in Chapter 6, “Concrete interaction”.

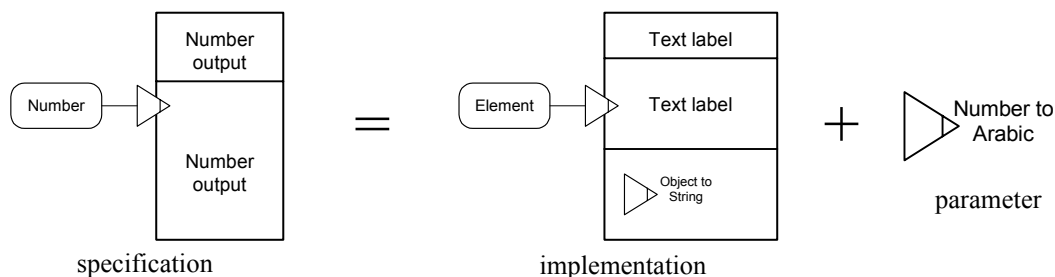
# Chapter 6

## Concrete interaction

### 6.1 Introduction

In Chapter 5, “Dialogue modelling”, an abstract model of interaction was presented, with the aim of providing a modelling approach that is independent of specific interaction styles and end-user platforms. Although (proto)typical interaction objects for some basic interactor were introduced and examples of such were given, there was little reference to specific graphical user interface elements like buttons, menus or windows. The concrete interaction objects were modelled as interactors containing gates and simple state machines, with little regard to the capabilities and requirements of their underlying toolkit counterpart. While it is important to be able to describe and analyse abstract characteristics of elements of the user interface, we still need to implement the abstract elements and structures in terms of the concrete platform and interaction style used for the final user interface, i.e. the elements that the user interface is eventually made of. Hence, we need to understand how the concrete interaction objects may be combined within a certain interaction style and the limits or requirements they put on composition.

The abstract dialogue represented by an interactor model can be seen as a specification for an implementation in terms of specific interaction objects. When going from the abstract specification to the concrete implementation, we take advantage of the capabilities of the interaction objects, to compose an interface with the desired functionality. For instance, to present a number to the user, we may combine a function from numbers to strings with the ability of text label objects to present strings. This is an example of using an interaction object for information output, and corresponds to the NUMBER OUTPUT interactor shown left in Figure 93. However, to be able to identify the text label interaction object as a way of implementing the NUMBER OUTPUT interactor, we need a description of the capabilities of the text label object, that helps us understand what is required for using it. In this case, we could model the text label as shown middle in Figure 93, to express that it can present any object given an appropriate function for making a string from it. In the case of number output, we could supply a function that uses the arabic notation for the number, as suggested right in the figure. In a sense, the *specification* equals the sum of the text label *implementation* and the function *parameter*, as indicated by Figure 93.



**Figure 93.** Using a text label for presenting numbers

The model of the text label in the middle is not necessarily a complete description of the capabilities of the text label object. It is one of many possible abstractions of the text label, and this particular one is suitable for identifying it as a candidate for presentation of objects. The function parameter is an example of an object that must be provided by the context that the text label is used in, and this is important to include in the model. The text label will usually also require a window container and this window must be a frame or be contained in a frame. These requirements should also be included in the text label model, to ensure that the text label interaction object is provided with an appropriate context to operate within.

In the figure we have used the DiaMODL language presented in Chapter 5, “Dialogue modelling” for modelling both the abstract specification (left) and the concrete interaction object (middle). For the purpose of this chapter, the difference between abstract dialogue and concrete interaction is not the language concepts or notation used, but rather the focus of the model. For modelling concrete interaction, the focus is on capturing the specific behaviour of concrete interaction objects and what is required for using them for implementing the specified abstract dialogue.

In this chapter we will show how different concrete user interface elements and interaction styles can be modelled. We will initially focus our discussion on standard graphical user interface elements, as found in most desktop interfaces and in the forms-oriented interaction style. Later, we will look at direct manipulation and mouse gestures, and show how the same modelling principles can be used to model this interaction style. Since the focus is on how these can be utilised for implementing the abstract dialogue specification, we will not discuss issues like perception and motoric requirements. For modelling we will use the same languages as used in Chapter 5, “Dialogue modelling”. RML is used for modelling interaction object concepts, while DiaMODL is used for modelling structure and behaviour.

## 6.2 Window-based interaction objects

Different types of interaction objects have different capabilities and each type will typically be oriented towards a particular role in the implementation of the abstract dialogue. Since we are interested in how different concrete interaction objects support the implementation of abstract interactors, it is natural to classify them in terms of the same three aspects as in Section 5.1 on page 83, i.e. information mediation, control and activation, and compositional structure. Any interaction object will typically have features of several of these, but



still have a distinct focus on one of them. For instance, although a push-button provides information in its text label, its primary use is that of controlling other parts of the interface, by means of the transitions between the pressed and released states of the button. Similarly, although some windows provide information through its title that may reflect the contents, the main role of a window is that of containing other interaction objects.

The information objects that an interactor supports input and output of, are usually domain data that are described in the task model. However, not only the domain data, but also meta-data may be manipulated in the user interface. For instance, a table may have column headers referring to attributes names from the domain model, as well as showing the attribute values of a set of elements. In fact, all the different user interface models may be the source of input and output information:<sup>1</sup>

- The domain model is used to indicate what parts of the domain data that is presented. For instance, in fill-in forms, labels are typically used for presenting attribute names from the domain model, left of or above the actual attribute values. In list views for large sets of values, the user may be allowed to filter out values, by selecting one of a set of concept specialisations that are defined in the domain model. In tables support for sorting is often provided by letting the user click on column headings containing attribute names, again part of the domain model.
- The action names used in task model are often used for labelling action-oriented elements like buttons and menu items. Toolbars often group tools according to the task structure, and “wizards” will typically support tasks that have been identified as crucial during task analysis.
- The two solution-oriented models, i.e. dialogue and concrete interaction, may be used for customising the interface, e.g. for tailoring views, toolbars and keyboard shortcuts.

The RML language introduced in Chapter 4, will be used for classifying concrete interaction objects and modelling their static structure. In particular, the aggregation construct/part-of relation will be used for modelling the containment structure. The information mediation role will be modelled using the interactor modelling constructs interactor, gates and functions, and the required containers will show up as interactor resources. The control role will be modelled using Statecharts constructs, and the alternative interpretation of aggregation as composition of dynamic state will be used for suggesting the Statecharts structure.

The model in Figure 94 shows the main abstract interaction object classes, based on distinguishing between different roles in the container-part structure. Looking down the hierarchy we distinguish between those that do (CONTAINER) or do not (SIMPLE) contain other DIALOGUE ELEMENTS. Looking up we distinguish between those that are (PART) or are not (FRAME) contained in another DIALOGUE ELEMENT. We see that a PART may be ENABLED or DISABLED, corresponding to two states, as shown in the right diagram. Although not expressible in RML, the DISABLED state means that the dialogue element *and* its parts ignore user actions like mouse clicks. This container-part structure is most evident in win-

---

1. It can be argued that a user interface should not only know the underlying domain, but also its own functionality and behavior, e.g. to support interactive help. This represents a kind of self-reference that may complicate the interface and presents an extra challenge for the designer.

dow-based user interfaces, since it usually maps directly to the spatial structure. However, other interaction styles can be interpreted in terms of these classes, e.g. by temporal sequencing.

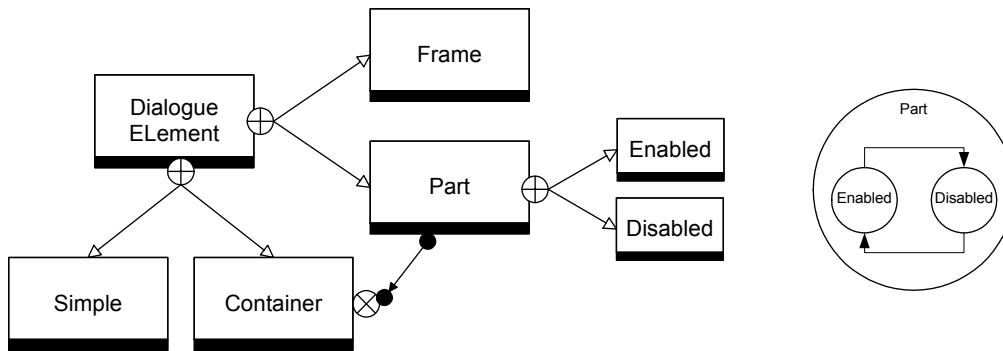


Figure 94. Dialogue container and element parts

Specific classes of CONTAINERS may limit the kind of PARTS they may contain and certain kinds of PARTS may only be contained in certain CONTAINERS. In Figure 95 we have modelled the special CONTAINER called FRAME that is used as top-level window of most applications. This kind of window usually provides means of minimizing, maximizing or returning to the normal state, and this is expressed right in the figure using a selection transition leading to corresponding states. According to the model, a FRAME is a CONTAINER that may contain a MENUBAR and a TOOLBAR, and must contain one or more PANES. In addition, MENUBARS, TOOLBARS and PANES, must be contained in FRAMES, and cannot be contained in other kinds of CONTAINERS. Since this entails that a FRAME cannot directly contain *other* DIALOGUE ELEMENTS than MENUS, TOOLBARS and PANES, other DIALOGUE ELEMENTS must be placed within one of the FRAME's PANES. Such constraints may both be due to technical limitations built into toolkits or to design conventions. It is often technically impossible to include MENUS in other CONTAINERS than FRAMES, and while TOOLBARS may technically be PANES, they should be attached to the border of FRAMES, and hence must be immediately contained within them. Since our goal of modelling concrete dialogue is to support design, it may be desirable to include both kinds of constraints, technical and design rules, in our model.

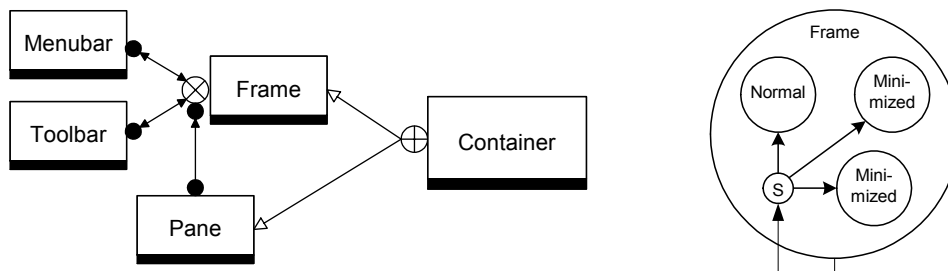


Figure 95. Frame and pane containers

The overall goal is to capture the meaning of concrete dialogue composition in general, to support composition of interactors. The model in Figure 94 is a start, since it defines the basic containment structure. As discussed and shown in Figure 95, additional composition constraints may be given. However, there are other attributes of DIALOGUE ELEMENTS that constrain composition, in particular their location, size and layout. While size is an attribute of an element alone, location and layout are relative to other elements, and to fully model

this, we need to introduce concepts like bounding boxes, an algebra for boxes and classes of layout strategies. The right state diagram in Figure 95, models additional size-related behaviour of FRAMES, where three states, NORMAL, MINIMIZED and MAXIMIZED are distinguished. These states all reflect a certain (strategy for determining the) size, and can be utilised when composing DIALOGUE ELEMENTS. Similarly, many DIALOGUE ELEMENTS provide various ways of gracefully adapting to space constraints, such as label truncation and scrolling. Although all these concepts complicate dialogue composition, they provides fairly good support for reasoning about composition, and hence support for interactor composition.

## 6.3 Simple interaction objects

In this section we take a look at the standard simple interaction objects typically found in GUI builders, toolkits and object-oriented frameworks.

### 6.3.1 Labels and icons

The text label concrete interaction object is used for most short and static texts in a user interface, like attribute names that proceed text input fields. One possible model of the text label interaction object is shown in Figure 96. According to this model, a text label consists of a 1) label string, 2) a font (specification) including the font family, size and **bold** and *italic* flags, and 3) two colours, one for the text itself and one for the background. This model is richer than the example in Figure 93, and is a better approximation of modern toolkit implementations, like Java's JLabel swing class.

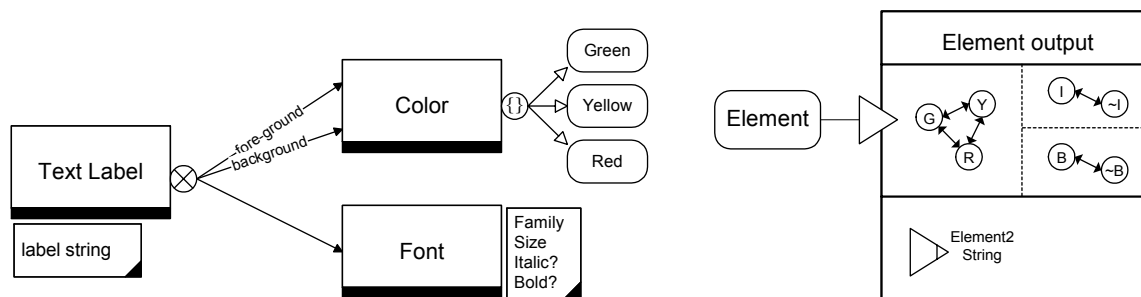


Figure 96. The text label concept and interactor

In most cases, the label string itself carries most of the information for the user, and has a dual use. First, it represents some datum in the domain of discourse, e.g. a name, date or numeric value. Second, it may refer to elements of the underlying model, e.g. concept/class, relation and attribute names, as an aid for making the interface easier to understand. The latter use is interesting since it highlights the user interface model as a source of output information, as discussed in Section 6.2.

While the label string is considered the main part of the text label, the text label's relations and attributes represent a considerable state space that can be utilised for e.g. classification purposes. The interactor model at the right illustrates this point, where three colours and

each of the two boolean attributes are represented as and-composed states. The transitions may e.g. be controlled by how the output/receive ELEMENT value is classified into specialisations of a general concept, where each group of states may represent one disjoint specialisation. Consider a three-axis classification of a process state, into either Normal, Abnormal or Critical, either Cold or Warm, and finally whether the rate is Increasing or Decreasing. If quantitative measurements are represented by the label string, the class membership may be mapped to each of the state groups, with Normal, Abnormal or Critical mapped to the three colours, Cold or Warm mapped to bold on/off and Increasing or Decreasing to italic on/off. In an even more extreme model, the colours would be modelled by three orthogonal colour components, which each representing one class membership. The point is not that this is a good design choice, but rather that even the simple text label interaction object provides a state space augmenting the label string, that may be fruitfully utilised when implementing an abstract interactor.

A similar reasoning can be applied to sets of related icons. Each icon may be modelled as an or-composition of colour states, like for text labels. A set of related icons may similarly be or-composed with each other, and finally and-composed with the composite colour state, giving a total state space of #icons times #colours. Icons are often combined with labels, and it might seem that the label discussed above could be (and-)composed with an icon, to give a total state space of #label states times #icon states. However, when composing concrete dialogues and state spaces with “physical” attributes like bounding boxes and colours, we should consider whether *merging* would make more sense:

- If the two parts, the label and icon, are supposed to represent one element in the domain, it seems reasonable to always use the same (background) colour
- Alternatively, if the label and icon represent independent domain elements, they should themselves be independent, and an and-composition seems reasonable.

This example shows that composition of concrete interaction objects is not as simple as the abstract model suggests, and that the concrete and “physical” aspects of the elements provide natural constraints for composition that must be considered.

### 6.3.2 Buttons

While text labels and icons are used for presentation only, buttons are interactive and can be used for input. The basic behaviour of buttons is the switching between the released and pressed states. When used for *control* purposes, the transition is typically triggered by a hardware button, e.g. left mouse button, the return key or a special physical knob. The two states may also be used for information mediation, where the pressed and released states are mapped to the boolean values true and false, respectively, e.g. based on an attribute value, element classification or the presence of a relation.

To understand the basis for the functionality of buttons, we again start with a model of the static structure, as shown left in Figure 97. A BUTTON contains either an ICON or a TEXT LABEL, as well as a border of which there are (at least) three kinds. This means that for presentation purposes, the BUTTON’s maximum state space is the states of the border and-composed with an or-composition of the ICON and TEXT LABEL, i.e. a considerable state space. At a minimum, the BUTTON needs two states to distinguish between the two boolean values,

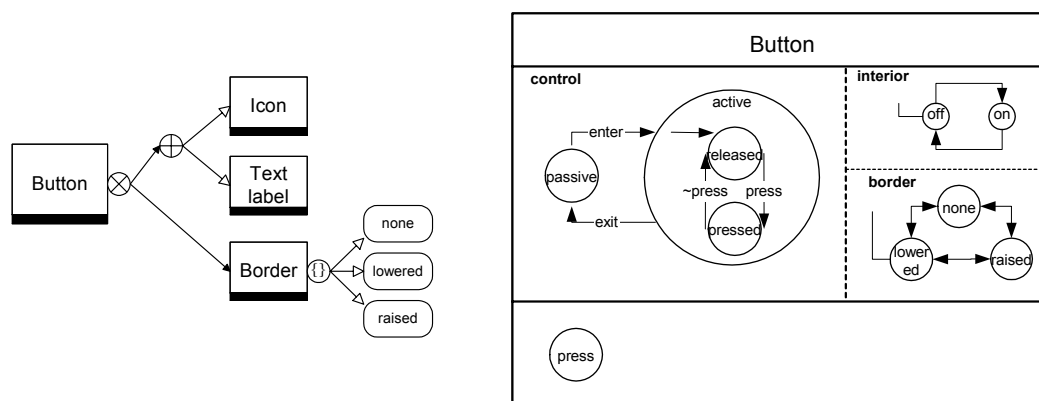


Figure 97. Button structure and states

while in practice, some additional intermediate states are used for user feedback. The right model in Figure 97 shows one possibility, including the presentation-oriented states INTERIOR and BORDER. These are driven by the CONTROL states, which limit the allowed state sequences to those providing the desired interactive behaviour. The PRESS state/condition resource in turn drives the CONTROL state and provides the link from some external state, e.g. a mouse, keyboard or other physical button. In the typical button implementation, the PRESS resource is bound to left-button and space-bar keystroke, the latter requiring that the button has the keyboard focus. The typical state sequence is INACTIVE, ACTIVE.RELEASED, ACTIVE.PRESSED, ACTIVE.RELEASED and INACTIVE. This corresponds to moving the mouse pointer over the button element, pressing and releasing the mouse button and moving the pointer away. By adding appropriate events and conditions to the transitions in the CONTENT and BORDER states, variants of the basic look & feel may be implemented.

Two basic look & feel variants of the button are normally provided. In the first variant, the border is instantly raised after the button is pressed, returning to the released state. This variant is mainly used for control, by triggering other transitions when reentering the ACTIVE.RELEASED state. The pressing of the button can be seen as an utterance or act, e.g. “Save”, directed towards the application or the context of the button, e.g. the current document. In the other variant, the button represents a boolean value that is toggled/inverted when the button is released. The button will remain in this new state until either pressed again or manipulated by the application. The button in this case reflects a state of the domain, e.g. the **boldness** attribute of a region of text, and pressing it means changing this state. If the context of the button is changed e.g. by moving the text cursor or selecting a different region of text<sup>1</sup>, the button must be updated to reflect the state of the new context.

1. In a text field or editor, a normal text cursor is normally interpreted as an empty selection.

The interpretation of these two variants as either control- or information-oriented, should be reflected in the corresponding interactor model. The control case is shown right in Figure 97, while the information interpretation can be indicated by adding output/receive and input/send gates. The input/send gate's tip must be connected using a complement connection to its own base, as shown in Figure 98. Table 7 shows the state correspondences for one observed implementation of this variant.<sup>1</sup>

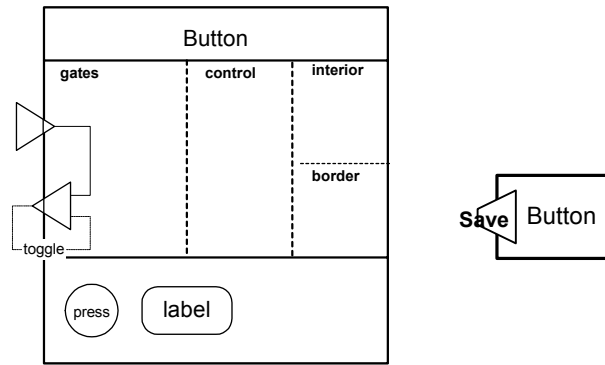


Figure 98. Boolean toggle (left) and act button (right)

	<b>inactive</b>	<b>active.released</b>	<b>active.pressed</b>
<b>true</b>	interior.off border.lowered	interior.off border.lowered	interior.on border.lowered
<b>false</b>	interior.off border.raised	interior.off border.raised	interior.on border.lowered

Table 7. Button control and presentation state correspondences

This classification as either control- or information-oriented button can however be misleading. As an act, the control case should be similar to speaking/uttering the button label, which is better modelled as input of the information contained in the utterance. Hence, in the case of the “Save” button, we should model this by adding an input/send gate emitting the “Save” act, as shown in Figure 98, and receive this act value some other place in the dialogue model, as discussed in Section 5.6. The information-oriented model of the **boldness** button may similarly be misleading, since both computing the current state and changing it may be complex operations, requiring the creation and modification of many elements. For instance, setting the **boldness** of the two middle words of “this partly *italic string*” to true, may require both creating a **bold** region for “partly”, a **bold italic** region for “*italic*” and shrinking the existing *italic* region to only contain “*string*”.<sup>2</sup> Hence, pressing the button should be considered a considerable act.

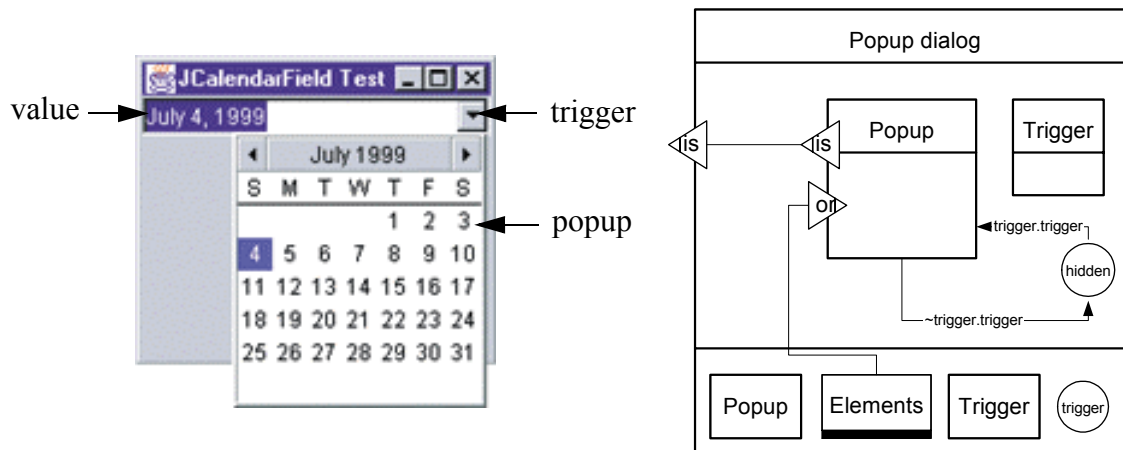
### 6.3.3 Popups dialogs, dropdown listboxes and menus

Menus and dropdown listboxes are special purpose composite interactors that implement a neat way of quickly accessing functionality without using too much screen space. Menus are typically used for providing access to the main application functions and are often hierarchical, i.e. the menu items are themselves menus. Dropdown listboxes are used for selecting one of many values, where the set of possible values may be dynamic and large. Both

1. Visio Professional 5.0 on Windows 2000

2. A *region* is a (sub)sequence of characters with specific attributes, and are easier to handle if they are non-overlapping. Hence, setting attributes of arbitrary regions may require creating, merging and deleting regions.

menus and dropdown listboxes are instances of the more general *popup* dialogue, where a trigger is used for temporarily activating another dialogue. Figure 99 (left) shows how a date field may be set by popping up (or dropping down) a complex calendar dialogue. This date interactor consists of a value part, which is itself an editable text field, a trigger and the popup dialogue, shown here in the active state.



**Figure 99.** A popup calendar.  
 • *Left:* concrete interaction object  
 • *Right:* generalised interactor model

Both menus and the dropdown listboxes are compositions of a trigger part and a selection part, where interacting with the trigger makes the selection part ready for accepting a selection, as shown in Figure 100. Although their look & feels are similar, the interactor signature of dropdown listboxes and menus are different, since the former is used for inputting data and the other for invoking actions. The model in the figure corresponds to the dropdown listbox, while to model a menu, the items would be a set of acts, and the selected act would be invoked, i.e. the input/send gate would be an act invocation. This is an example of the difference between dialogue and concrete interaction: The latter is the implementation of the former and similar implementations may correspond to very different functionalities. The similarity unfortunately makes it easy to abuse these two element types, e.g. use dropdown listboxes for act invocation. Note that according to the interactor signature, dropdown listboxes can be used to select an act element without invoking it, e.g. to specify an act as an element for later invocation by a higher-order function or scripting machinery.<sup>1</sup>

The POPUP DIALOG interactor model shown right in Figure 99 is a generalised variant of the POPUP SELECTION interactor, i.e. it represents a generic popup dialogue. The interactor is heavily parameterised, including the popup and trigger interactors as resources, as well as resources that relate to these two sub-interactors. By providing calendar and arrow button interaction object parameters, in addition to a set of months and a triggering state, the popup calendar shown left in Figure 99 may be released.

The way that the menu trigger activates a hidden dialogue, is similar to other popup dialogues, like Open... and Save... dialogues, but there is one important difference: The func-

1. As discussed in Section 5.6 and above, the difference between selecting a new and different value for an attribute and invoking an act for setting the attribute, is not that clear-cut. The main point is that the user doesn't expect dramatic things to happen when operating a dropdown listbox, while invoking an act is a more explicit request with potentially dramatic consequences.

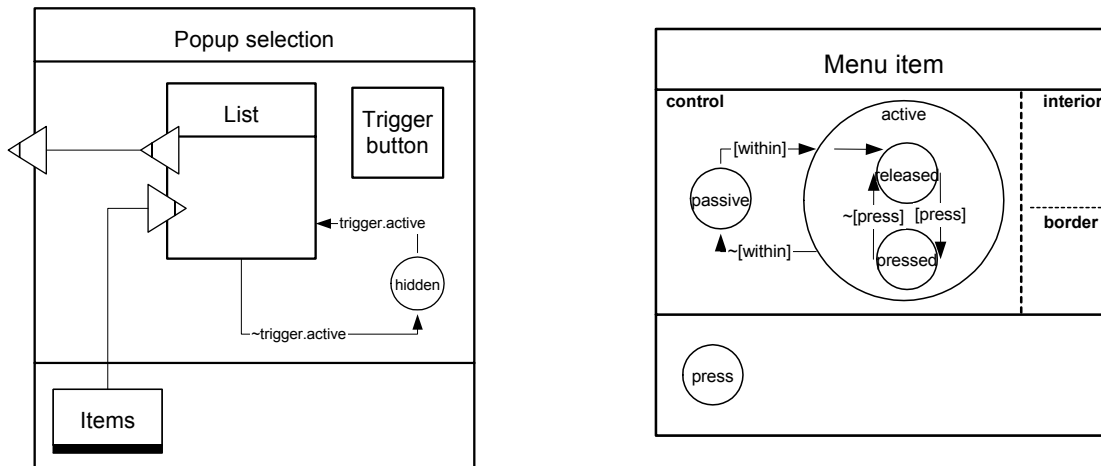


Figure 100. Popup selection and menu item interactors

tionality of menu items is limited, since it must be possible to both trigger and operate them using a single device or event, typically a mouse button press, move and release. This means that in general, menu items essentially are various kinds of buttons, e.g. boolean buttons, act buttons and sub-menu triggers. The model in Figure 98 includes this triggering event explicitly in the form of a state resource.

Thus, a menu represents a composition of a trigger button and a set of item buttons, where both control and layout is coordinated. The composition of the control aspect must ensure that the trigger button activates the set of item buttons and that these reacts in a coordinated manner. The layout must support the presentation of a uniform set of elements, e.g. a one- or two-dimensional grid. However, the button model shown in Figure 98 cannot be used directly for menu items, because of the way they are activated during a mouse drag. Figure 100 shows an alternative menu item model, where conditions are used for triggering transitions, rather than events. As can be seen, the button model will enter the `RELEASED` state only when the `press` event occurs within the button, while a menu item will enter the `RELEASED` state whenever the `within` and `press` conditions are both true. Hence, dragging the mouse across several menu items will activate the `pressed` state of each item in turn.

Some menus support a mode of interaction, where the selection part remains open if the mouse button is released while still within the trigger. The selection may then be operated while the mouse button is released, with the menu items being activated as if the `press` condition still was true. This leads to a second difference between buttons and menu items: The menu item must give the same feedback for the active state as the button does in the `pressed` state. Alternatively, the `press` state resource must remain true until the mouse button is released *outside* the trigger. A similar behaviour may be observed for the set of top level trigger buttons in the menu bar: Once one trigger is activated by a `press`, the others will become activated just by passing the mouse button over them.

As shown in the figure, each menu item emits an `act` event, which is the result of interacting with the menu as a whole. In case of checked menu items, this `act` must toggle the state of the corresponding item button, while for dropdown menus, the menu trigger label should be set to the selected value.



The presented menu model is not be a complete and precise model of real-world menus, e.g. it does not take into account keyboard navigation. However, it illustrates several important points:

- Special purpose interactors may be modelled as tightly coupled simpler interactors, as exemplified by the trigger and item buttons.
- Composition if interactors requires careful consideration of how each part contributes to the overall behaviour, as illustrated by how menu items differed from buttons.
- Abstracting away the details of the parts, may prevent the integration needed for achieving the desired behaviour. For instance, the interactor signature of buttons was too abstract for the purpose of using them as menu items, as we needed access to their inner states.

Menus have a special role in desktop user interfaces, as it the main repository of predefined acts. The menubar is the main container for menus, which may contain either acts (buttons) or other sub-menus. As Figure 101 shows, there are few constraints for how the menu hierarchy is composed. Hence, it is easy to add or remove acts from this kind of container. In practice, there are two limiting factors: 1) The user will have problems navigating in deep structure or menu with many entries, and 2) there are conventions for naming and positioning menus, as well as populating them with acts.

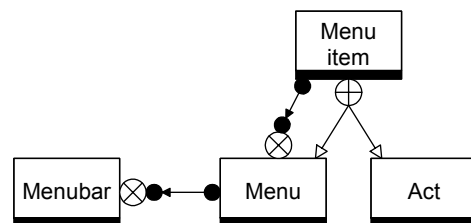


Figure 101. Menu aggregation/part structure

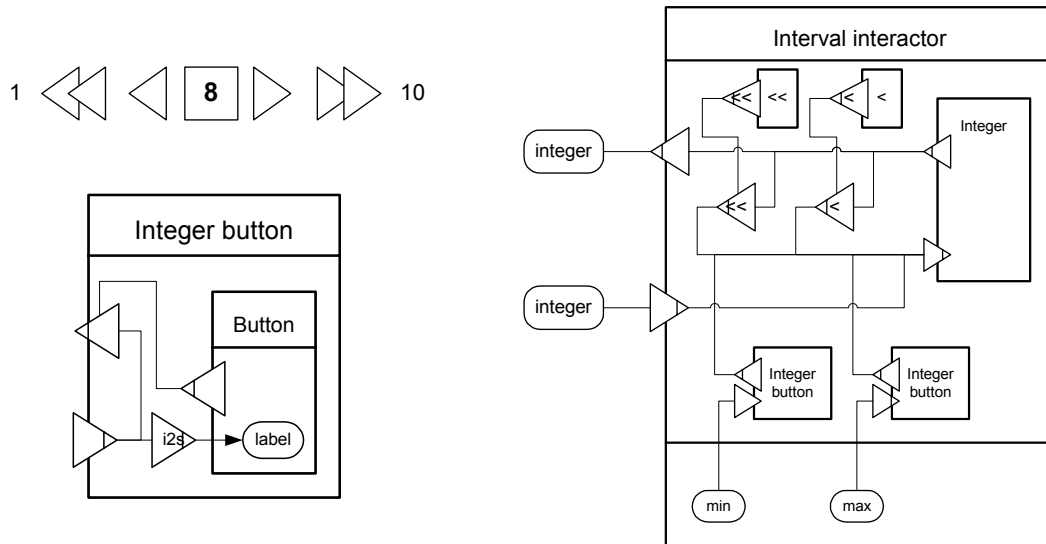
## 6.4 Composite interaction objects

Dialogues may be composed in several ways, depending on space and time requirements for the new dialogue. The dropdown listbox discussed above is an example of an element designed for using less space, but trades ease and speed of operation by introducing the triggering step for activating the main interaction object. It is expected that different user interface and interaction styles support composition in different ways and according to different formal and informal rules. According to the abstract dialogue/interactor model, any interactor has properties related to information, control and structure. Hence, composition of interactors may be discussed in terms of how the information, control and structure properties are derived from its the corresponding properties of the parts. By analysing typical interactor structures for a particular interaction style, heuristics for composition may be established.

### 6.4.1 Composition of function

The composition's *functionality*, i.e. interactor gate and resource signature, may be more or less directly derived from all or some of its parts. In some compositions, most of the parts

are used for controlling or supporting a main interaction object, the gates of which are connected to the composition's gates. Figure 102 shows how a specialised integer device may be composed from smaller parts. The concrete interaction object is shown top left. Recall from Section 5.2.1 that a device is an interactor with only input/send and output/receive gates, with the same type. I.e. an integer device both outputs an integer and lets the user input one.



**Figure 102.** Integer (within interval) device

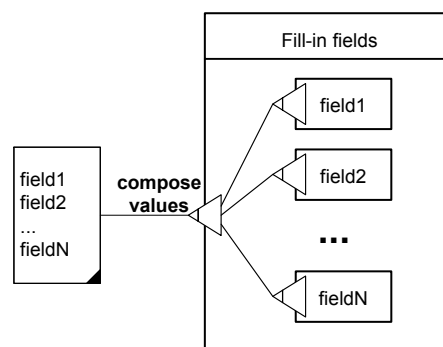
- *Top left:* concrete interaction object
- *Bottom left:* Integer button interactor
- *Right:* Composed integer device

The integer device is built around a main integer interaction object, in the example shown as an integer text field holding the number 8. As shown in the model at the right, its output/receive and input/send values are also the values of the composition's output/receive and input/send gates, and all the other elements support various ways of changing this value. The composition supports three ways of changing the value in each direction. Two INTEGER BUTTON interactors are used for setting it to the minimum (1) and maximum (10) values, while two pairs of arrow buttons support single- and multi-stepping in each direction. The integer button implementation is shown bottom left in the figure. The output/receive value is used to set the button label, and the same value is emitted through the input/send gate when the button is activated. The two pairs of arrow buttons are modelled as act input devices (only one pair is shown), named “<<” and “<” for single- and multi-step. I.e. they emit act elements, which in the right model in Figure 102 are used to trigger corresponding functions which compute the new integer value.

The composition of information flow is shown right in Figure 102. The output/receive and input/send values of the composition is fed in and out of the inner INTEGER text field interactor. The input/send value is in addition fed through two stepping functions, the action of which are triggered by corresponding buttons (only two of the four arrow buttons are shown). The two INTEGER BUTTON interactors at the bottom, directly set the output/receive value of the main INTEGER interactor to the minimum and maximum values that are provided as interactor resources. In this particular example, there is only one main value interactor, the value of which is manipulated by the other parts. Such a composition is typical for

values with a limited domain or a small set of relevant movements within the value space, like stepping and jumping to the extreme values, as in this case.

In other compositions, each interaction object contribute one of many essential data values, which are combined into a composite value. The typical case is a record or object with a set of fields that must be filled in by the user, like a date composed of year, month and day. An interactor for a record value will be composed of interactors for each field, as illustrated in Figure 103, e.g. interactors for year, month and day parts in the date value case. As shown in Figure 103, the input/send values of each sub-interactor would be connected to the input/send gate of the composition, which would construct the composite value from the parts using an appropriate function, before emitting it. The output/receive values of each sub-interactor would similarly be connected to the output/receive gate of the composition, and each connection would include a function for extracting the field's value (this is not shown in the figure). These two styles of composition may of course be combined. For instance, the integer interactor shown in Figure 102, could with appropriate minimum and maximum values, be used for all the parts of the date interactor.

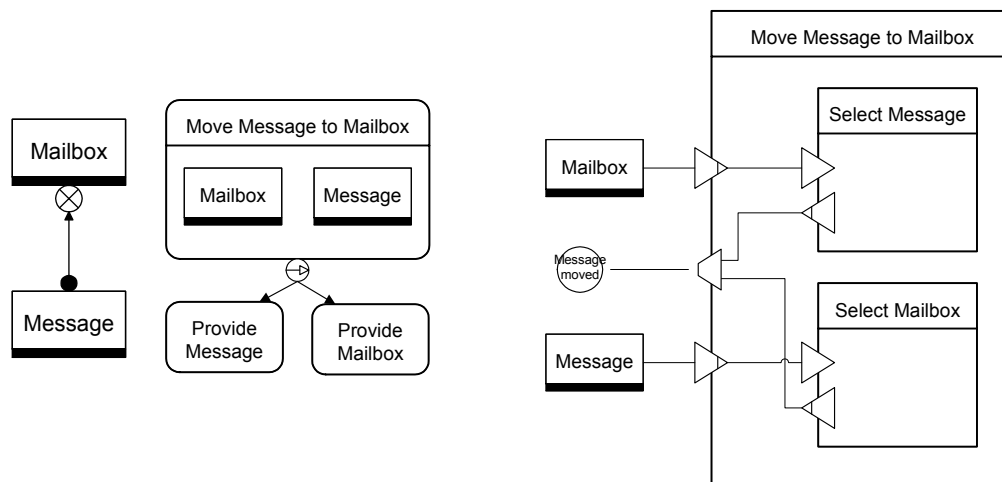


**Figure 103.** Composing an input device for a set of fields

By including sub-interactors of an interactor as parameters in the resource bar, we may allow the context where the interactor is used to customise the interactor's appearance and behaviour. For instance, to allow presenting the minimum and maximum values of INTEGER INTERACTOR in Figure 102, as roman numerals, we could include both the INTEGER BUTTON sub-interactors as parameters in the resource bar of INTEGER INTERACTOR. When using INTEGER INTERACTOR in a new context, we may bind these parameters to interactors that output integers as roman numerals instead of arabic. A similar technique is used by the Java Swing library, to allow a customisation of composite interaction objects. For instance, the JTree component, which is used for presenting hierarchical object structures, may be given a *renderer* component that is responsible for outputting each element in the hierarchy.<sup>1</sup> The JTree implementation will give this renderer each visible object in turn, and ask it to paint the object at the appropriate position. If no renderer object is explicitly provided, a standard JLabel component is used. The interactor corresponding to the JTree interaction object is shown in Figure 104. As in Figure 66, Section 5.2.3, the model expresses that the interactor may output hierarchical structures and support the selection of a leaf node. In addition, a RENDERER parameter is included, which is bound to a JLABEL interactor by default. It is explicitly stated that the RENDERER must be able to output any node in the hierarchy. If editing was to be allowed, the Renderer parameter would in addition need to include an input/send gate. Although the model does not express exactly how the JTREE interactor uses the provided parameter internally, the model does express the essential capabilities that the parameter must provide.

1. Actually, a (different) renderer may be provided for each object in the hierarchy.





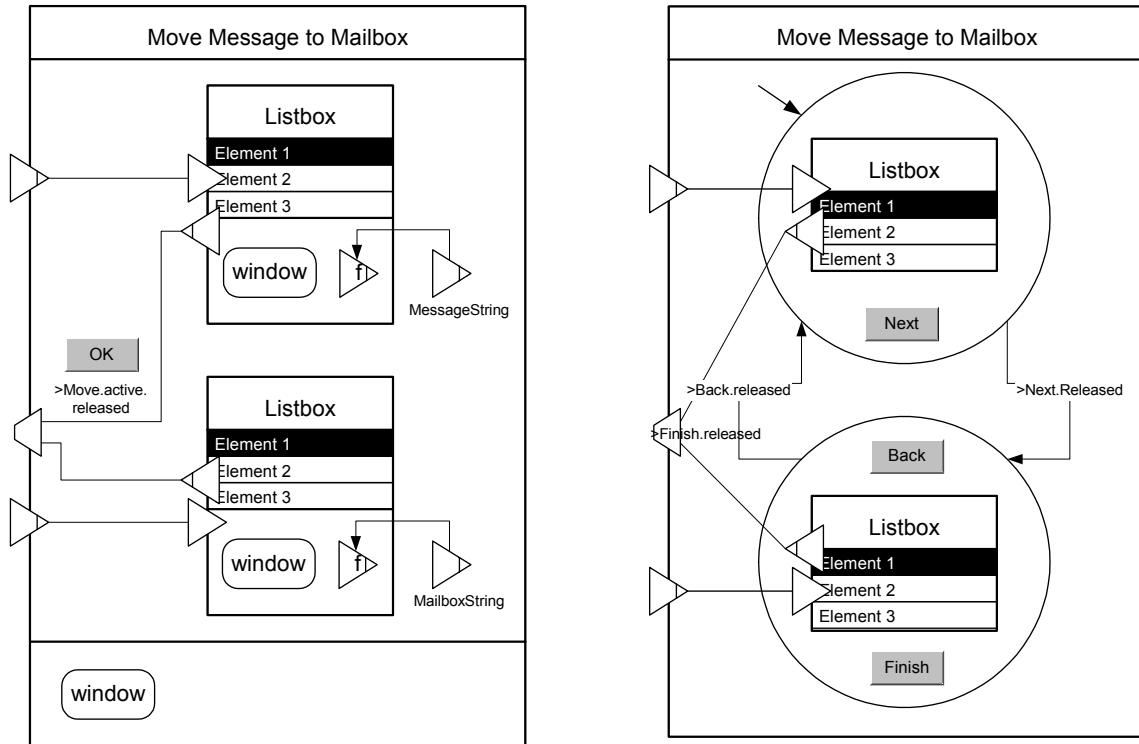
**Figure 105.** Task and dialogue model for moving messages among mailboxes

ate for the respective data types. The “OK” button is also shown in this figure, and is assumed to have a state named `ACTIVE.RELEASED` which is entered ( $>$ ) after pressing it, as shown in Figure 97.<sup>1</sup> Note that these three interaction objects are implicitly and-composed, since none are connected by transitions. A second more sequential design suggestion should also be considered. The two listboxes may be activated in turn before the act is triggered, and to avoid errors navigation back and forth between the listboxes may be supported. A model of such a dialogue is shown right in Figure 106. It includes the same two listboxes and adds buttons for navigating between the listboxes. Note that the information flow is unchanged, only the control/activation structure is different.

These two possibilities results from the underlying modelling concepts and constructs of the dialogue/interactor modelling language. For this to be meaningful in the concrete interaction perspective, both kinds of composition must be given a natural interpretation in terms of the concrete resources used by the interactor parts, in this case `WINDOWS` and `DIALOGUE ELEMENTS`. As shown in the RML model of these concepts in Figure 94, there is an aggregation relation between `WINDOWS` and `DIALOGUE ELEMENTS`, which naturally corresponds to parallel composition. Since the cardinality has no upper bound, it seems that we can compose an unbounded number of `DIALOGUE ELEMENTS` in a `WINDOW`. This may be a fair approximation for some window types, e.g. scrolling frames. However, a deeper model with concepts like bounding boxes, layout, scrolling and resizeability would reveal that different design targets have different practical limitations. In the case of small fixed-sized panes this could either be handled by using the same parallel composition with different smaller-sized interaction objects, e.g. dropdown listboxes, or moving to the alternative wizard design.

The sequential design is based on activating and deactivating the interactors, and assumes that the state interpretation is supported by the runtime, through dynamic manipulation of the part-of relation between the `WINDOW` and `DIALOGUE ELEMENTS`. In more practical terms, this means either removing `DIALOGUE ELEMENTS` from the `WINDOW`, or making them invisible, hence allowing several `DIALOGUE ELEMENTS` to share one `WINDOW`. An intermediate solution is actually possible, as the “Disabled” state of the `DIALOGUE ELEMENTS`, i.e. disa-

1. In the figure, the two selection interactors are shown as stereotyped listboxes with gates, and the act trigger as a push button. This is a useful hybrid notation which mixes abstract behavior with concrete interaction.



**Figure 106.** The two Listbox designs: *Left*: dialogue box, *right*: wizard style

bling instead of removing the dialogue, can give the same effect of sequential activation. However, this would violate the wizard style and give no reduction in space requirements.

Apart from the overall structure, there are two smaller differences between the left and right designs in Figure 106: 1) the use of “Next” and “Back” buttons for switching between states, and 2) the label used for the committing button, “OK” and “Finish” respectively. The first difference can be attributed to the structure, i.e. some event or condition is needed for triggering the two transitions. The second difference is due to the “wizard” interaction style which requires a clear indication of commitment.

### 6.4.3 Composition of structure

The third aspect of an interactor is the containment structure and the relation between interactor characteristics at different levels in the hierarchy. In the context of window-based interaction objects, this concerns how interactor resources are shared and divided between sibling interactors, based on the resources provided by the parent interactor. This is partly dependent on interactor and state machine composition, but is also highly dependent on the concrete interaction objects. In the example shown in Figure 106, both the listboxes require a containing window and this window is provided through the resources of the containing MOVE MESSAGE TO MAILBOX interactor. In the case of the left design suggestion, the listboxes are active at the same time and hence require non-overlapping window areas. In the right design suggestion, the listboxes are or-composed and mutually disjoint, and hence can share window space. In both cases, the button(s) must also be allocated space in the window. Again, the main principle is that and-composed interactors must have non-overlapping bounding boxes, while or-composed ones may share window space. There are however, exceptions to this rule. It is not uncommon that all dialogue components at one level in the

interactor hierarchy are laid out together, whether they are and- or or-composed. Activating or deactivating a component is either mapped to hiding/showing or dimming/highlighting it. This makes the layout more stable, since hiding or showing an element will no change the layout. Most modern object-oriented toolkits provide *layout managers* which are responsible for computing the position and dimensions for each component. In the example in Figure 102, all the sub-components occupy the same container, with a horizontal left-right layout. Such a simple layout may be formalised using two dimensional interval algebra [Mukerjee, 2001]. To model complex layout strategies including wrapping and standard layout tools like alignment and distribution, we must introduce side-effects, iteration over sets of boxes and conditions/guards.

In the example shown right in Figure 106, each interactor was implemented by distinct listboxes. However, since these two interactors were or-composed, they could actually be implemented by a single concrete listbox. This single listbox would be populated by different sets of data, depending on which interactor was active, and the current selection would flow out of the corresponding input/send gate. In this case, the MOVE MESSAGE TO MAILBOX interactor would require a listbox as a resource, and this listbox would be provided to the two selection interactors. In the general case, each interactor could require specific concrete interactors to handle the concrete interaction of themselves and their sub-interactors. High-level interactors would require larger dialogue structure, i.e. instances of models like those shown in Figure 94 and Figure 95, which would be divided or shared among their sub-interactors, depending on the stated resource requirements, compositional structure and control logic. The hierarchy of concrete interaction objects would provide a solution within the concrete interaction perspective, of the hierarchical specification of abstract interaction, as suggested by the framework shown in Figure 11 in Chapter 3, “Design and representation”.

## 6.5 Direct manipulation and mouse gestures






The previous sections showed how window-based interaction objects could be modelled and analysed in terms of composition of interactors. The correspondence between the abstract interactor model and the concrete interaction objects is quite easy to establish. The rules for composition are relatively simple, as they are based on the language primitives and 2-dimensional surface layout. In this section we put our focus on the richer interaction of direct manipulation and mouse gestures.

To introduce the various aspects of mouse gesture recognition in the context of direct manipulation, this section presents a mouse gesture example. This kind of mouse gesture is found in most desktop interfaces to file systems, and this particular variant is found in Microsoft’s Active Desktop™. Table 8 illustrates and explains the main steps of the gesture.

This example reveals several important points about gesture recognition in general:

- The gesture was sensitive to the file and folder icons, i.e. a gesture can depend on the set of *displayed elements*.

- The gesture recognised the relation between mouse pointer position and icon area, i.e. a gesture must be able to detect display element *hits*.
- The feedback included *changing* the cursor, *highlighting* displayed elements and *displaying* copies of displayed elements, i.e. a gesture must be able to give *visual feedback*.
- The gesture is completed by emitting an act, which is received by (an interface to) the operating system.

Step	User action	Feedback
	The mouse pointer starts outside both icons.	The mouse pointer is an arrow and no icon is highlighted.
	The user moves the mouse pointer over the file icon.	The mouse pointer changes from an arrow to a hand and the file icon is highlighted.
	The user presses the left mouse button and moves the mouse pointer towards the folder icon	The mouse pointer changes to an arrow and a dimmed (copy of the) file icon follows the mouse pointer.
	The user moves the mouse pointer over the folder icon.	The folder icon is highlighted.
	The user releases the left mouse button.	The feedback is removed, the Move file to folder command is executed and the desktop is updated.

**Table 8.** Move file to folder gesture

The Active Desktop™ interface support two related gestures used for opening files and applications and moving file icons. The relation between these three gestures, identify some further points:

- If the mouse button was released outside of the folder icon, the file *icon* would move and not the file itself, suggesting that a *different* command was issued.
- If the mouse button was released before having moved the file icon at all, a third command would be executed, suggesting that not only the release *target* mattered, but also the release *timing*.



We have chosen to model the meaning of gestures as acts or commands as used in CLIM [Rao, 1991], as opposed to a state or side-effect oriented one, as used in Garnet [Myers, 1990b]. In the former model the meaning of the gesture is only *suggested* by the feedback, the actual semantic interpretation is implemented by a function outside the gesture recogniser, which is activated when the gesture (recognition) is completed. This gives a cleaner separation of user interface functionality, and provides better support for e.g. macro recording and scripting.

The gesture example and the two variants described above, demonstrate three important points:

- The actual act that is issued is not necessarily determined until it is emitted, i.e. after the last gesture step.
- The act parameters can be a function of several aspects of the gesture, e.g., mouse position of button release, display elements that are hit when pressing or releasing the button or the objects that these represent.
- The actual gesture aspects that are used as act parameters, can be determined both at the same time as the act and not before.

The next section will introduce our approach to modelling the identified aspects and its role within an application and its user interface.

### 6.5.1 Structure of recognisers

The previous section identified several important features needed in a gesture recogniser, e.g. cursor feedback and highlighting, hit checks, commands and their parameters, mouse handling, presentation and display elements. Figure 107 shows an RML model of these aspects and elements.

The main features of Figure 107 are the hierarchy of GESTURES, the set of GESTURE parts and their relation to DISPLAYELEMENTS. A generic GESTURE has a set of STEPS and MOUSEGESTURE additionally includes a MOUSE, KEYBOARD, HITCHECK and FEEDBACK. DRAG'NDROP is a MOUSEGESTURE, for dragging from a source DISPLAYELEMENT to a target DISPLAYELEMENT. Our example gesture, MOVE FILE TO FOLDER, is a DRAG'NDROP gesture. The FEEDBACK part can use VISUALS for feedback, while HITCHECK will indicate which of the VISUALS it checks for hits on. The VISUAL concept provides the link to the underlying application data, and is either an ATTRIBUTE or ELEMENT visual. The represents relation is used to navigate from the visual layer to the underlying elements of the domain.

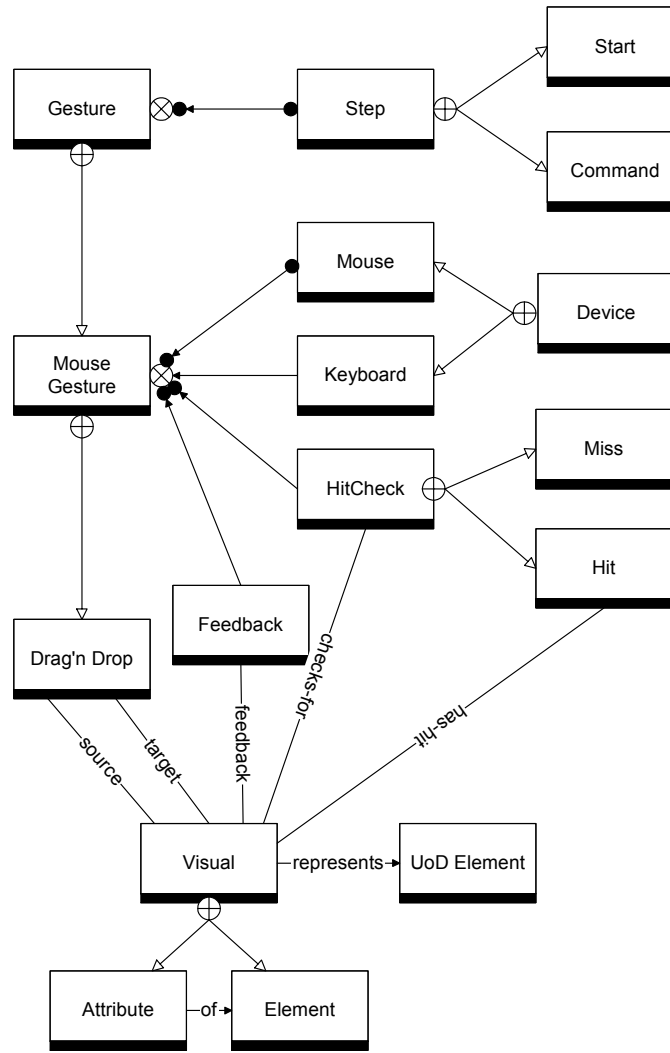


Figure 107. Aspects and elements of gesture recognisers

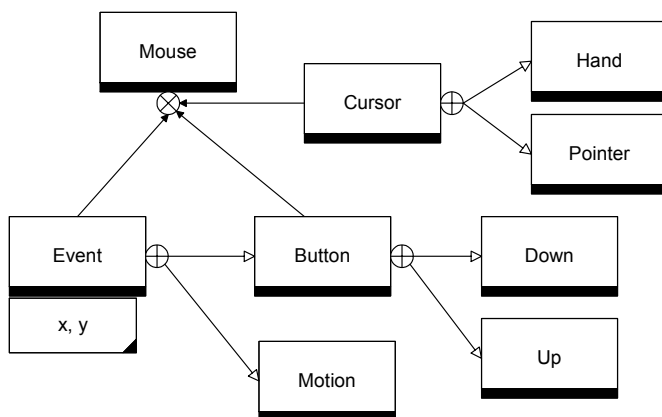


Figure 108. Mouse device

The MOUSE and KEYBOARD concepts refer to physical devices generating events and basic bits of information, that drive the gesture recognition machinery. Figure 108 expresses that MOUSE contains BUTTONS, which can be either up or down, two kinds of CURSORS, and finally EVENTS, which have x and y attributes for recording mouse coordinates. EVENTS are of two kinds, BUTTON and MOTION. The difference between this model and the one in Figure 76, is due to the richer interaction provided by direct manipulation. As with the static model of dialogue containers

and elements, these models can be interpreted as the state space within which the interaction model defines the behaviour.

For a user, the behaviour of a gesture recogniser is visible through the feedback generated during recognition of the primitive device events. In the table describing the gesture example, we listed a sequence of steps that were identified by observing the relation between user action and feedback. For our example, this amounts to five states, one for each step, including the initial start state and the command issuing step completing the gesture. The relation between user actions and feedback, i.e. feedback to the user actions, is modelled by adding transitions that are triggered and controlled by appropriate (device) events and conditions, respectively. An conceptual Referent model and a preliminary Statecharts model of the main steps are shown in Figure 109 and Figure 110.

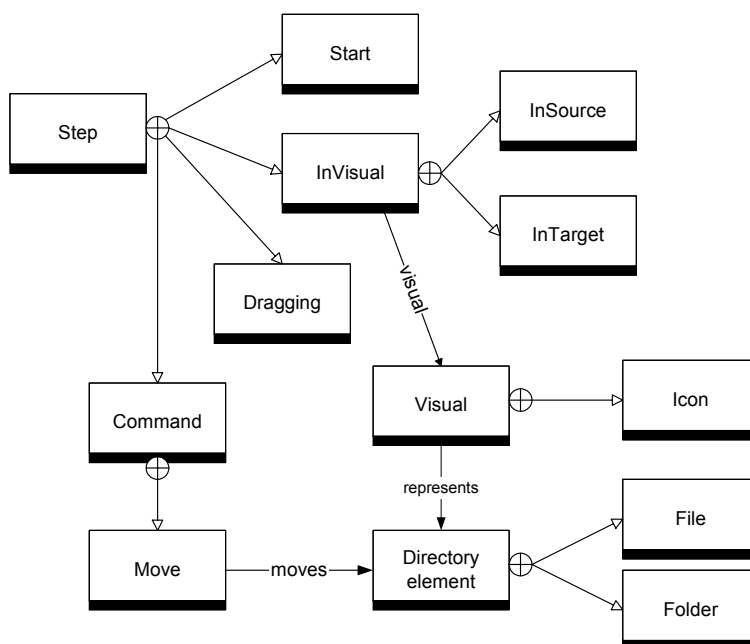


Figure 109. The structure of the five gesture steps

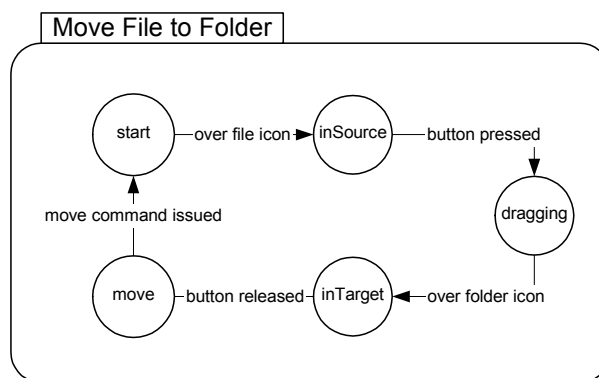


Figure 110. The behaviour of the five gesture steps

In the conceptual model, several special kinds of steps including the final move command are shown, in addition to part of the file system domain. In the state machine, the user actions performed with the mouse are only illustrated by the transitions. However, according to the previous section, the MOUSE referent and its events can be modelled as State-

charts states and transitions, and similarly for of hit checks and feedback. This will let us provide a more formal and complete representation of the behaviour, as shown in Figure 111. The notation “>state” and “<state” is short for “entering state” and “leaving state”, respectively. The “hit” condition is short for a check for if the mouse cursor is in/on one of the display elements that the hit state is related to through the checks-for relation.

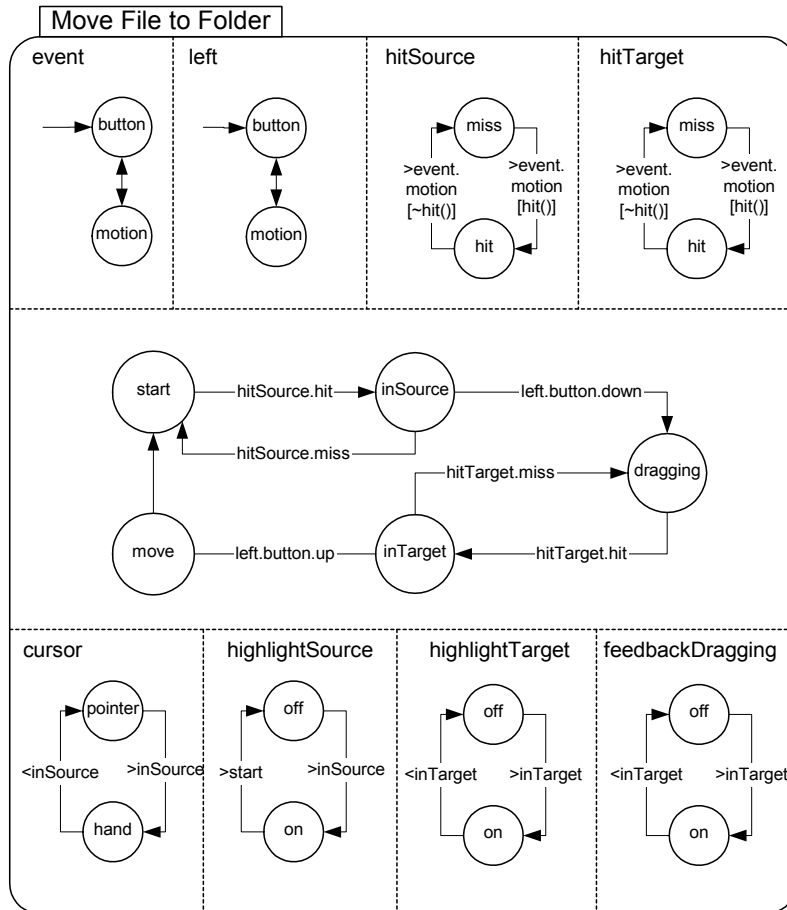


Figure 111. The Move File to Folder gesture recogniser

The states in the top row are concerned with user actions, the unconditional double transition illustrating that these are controlled from outside the Statecharts machinery. These top four states drive the transitions in the main state in the middle, which contains states for the steps of the gesture. The four states at the bottom are concerned with feedback, with one state for the cursor, two for highlighting and one for dragging icons. All these are driven by the main state, hence ensuring that the steps are clearly visible for the user.

An important aspect of the dynamic behaviour still remains to be specified, the relations between hit detection, feedback issued commands and the visual and domain objects these refer to. Some of these relations are represented as static relations and properties in the RML model. However, we also need to define when and how these relations and properties are established, changed and used, as part of the dynamic behaviour.

Among others, we need to specify:

- what display elements the hitCheck states should detect hits on

- how the feedback states get to know which display elements to highlight or drag
- to what values the parameters of issued command are assigned

To handle these aspects, we will take advantage of the fact that in our Statecharts dialect, states are also Referent elements, and hence can have relations and properties. The action part of the state transitions will be used to set these relations and properties.

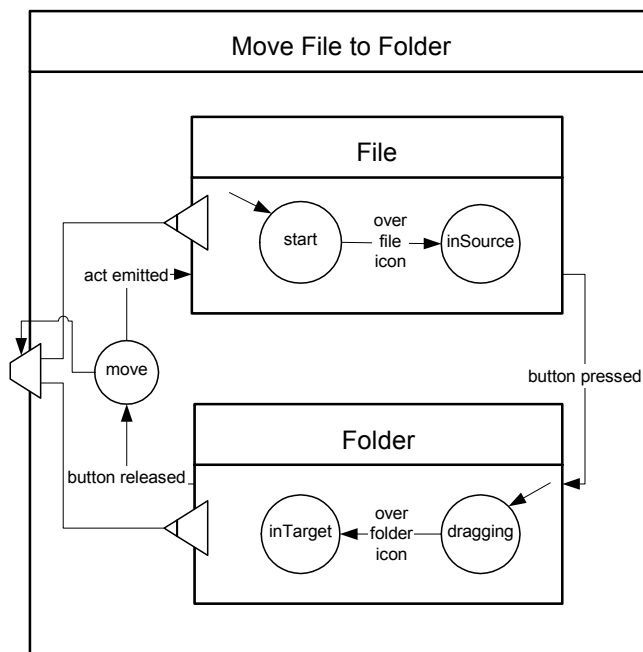
For some states, the relations and properties will be set by other components of the application architecture, before entering the states. In our example, the x and y properties of the button and motion event states are set by the devices, and the has-hit relation of the hitSource and hitTarget hit states are established by the presentation component. For other states, the information is computed from existing relations and properties, as part of the transitions' action part, as shown in Table 9.

State	Relation/Property	Value
hitSource	checks-for	Move File To Folder.source
hitTarget	checks-for	Move File To Folder.target
inSource	visual	hitSource.hit.has-hit
highlightSource.on	feedback	inSource.visual
feedbackDragging.on	feedback	inSource.visual
inTarget	visual	hitTarget.hit.has-hit
highlightTarget.on	feedback	inTarget.visual
move	file	inSource.visual.represents
move	folder	inTarget.visual.represents

**Table 9.** Transition actions

Compare the model in Figure 111 with the interactor model shown in Figure 112. The latter contains one interactor for each of the two arguments to the MOVE FILE TO FOLDER function, i.e. a file and a folder. This interactor represents a straight-forward design for supporting the Move File to Folder act, similar to the ones shown in Figure 106. Mapped to the main gesture steps, each of these sub-interactors corresponds to two of the steps, as shown in the figure, while the last step corresponds to invoking the act, i.e. applying the MOVE FILE TO FOLDER function to the file and folder arguments.

The interactor model can be seen as an abstraction of the gesture state machinery, focusing on the information aspect, rather than the control aspect. We see that the two sub-interactors provides one act argument each, in sequence, similar to model b) in Figure 106. Alternatively, the interactor may be seen as the functional specification for which this state machine is a solution in the direct manipulation interaction style. If the target was a wizard style, we would probably suggest a design similar to model b) in Figure 106. In fact, both the wizard style design and the direct manipulation gesture include basic states that can be grouped according to which arguments it supports the selection of. However, the wizard model is more hierarchical and simpler, while the gesture model is flatter and has a more complex



**Figure 112.** Move File to Folder interactor

control structure. It is possible to make the gesture model more similar in structure to the interactor and wizard models, e.g. if each icon is modelled as a button, we could compose a list for each icon type and get a model very similar to the wizard. However, the models have natural differences due to how different targets styles naturally require different models of concrete interaction. Some reasons for the differences are:

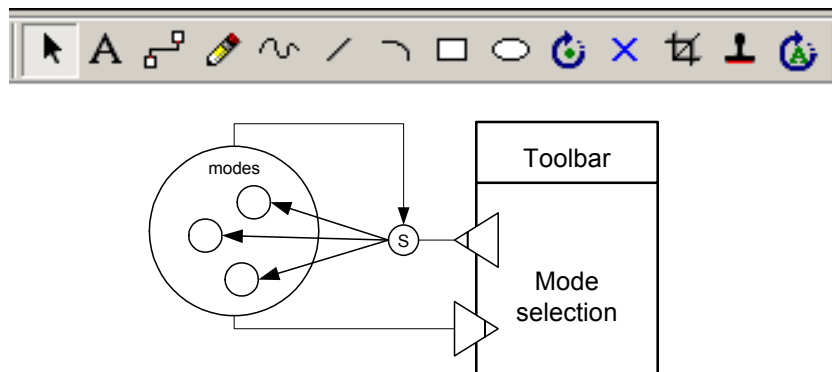
- The presentation structure is more complex for the gesture case, since the folder and file icons are spread out with no predefined layout
- The feedback for the gesture is richer and more complex

As noted, the standard interaction objects, e.g. the listboxes and buttons that were used in the wizard design, are easy to compose, since there are few constraints for how they can be grouped. They do not compete for other resources than screen space, and this problem is solved with a suitable layout. If more functionality needs to be integrated, e.g. another act must be supported, this would be fairly straight-forward. The complexity of the gesture machinery, on the other hand, makes composition of gestures more difficult. This can be seen in the model above, where the hierarchical structure of the interactor specification, was somehow lost in the gesture solution. If we needed to compose several other acts, like the mentioned MOVE FILE ICON, with MOVE FILE TO FOLDER, the complexity would grow and the design would be more difficult.

## 6.6 Composition of gestures

The standard way of composing gestures in a graphical editor or drawing application is through a toolbar or tools palette. Figure 113 shows an example of a toolbar and an interactor model describing its behaviour. The main idea behind such a toolbar is showing the available tools and letting the user select one for the duration of the next mouse gesture(s).

Each tool acts as a *mode*, since it determines how subsequent mouse presses, movements and releases will be interpreted. In the figure, each mode is modelled as a state and the set of available modes are or-composed into the MODES state. The TOOLBAR interactor is used for mode selection and its input/send value controls the mode or state that is entered through the selection (S) mechanism of Statecharts.

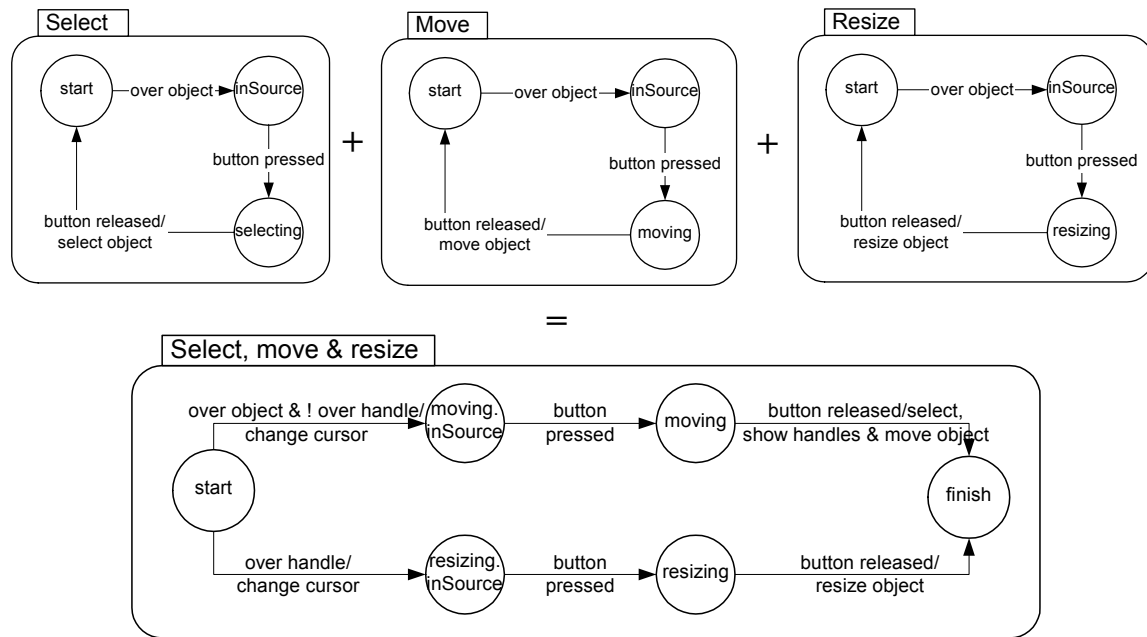


**Figure 113.** The Visio 5 drawing tools (top) and interactor model of toolbar functionality

The alternative and-composition of modes would result in all modes being active simultaneously, with severe effects. In the case of the Visio modes shown in Figure 113, the user would press the mouse and drag out a line, rectangle and circle at the same time. The problem is that the line, rectangle and circle drawing tools all react to the same mouse gestures, so the solution is force the selection of one mode to avoid the conflict. By letting the user freely select a different tool, all tools are in practice available, a kind of composition along the time axis. The advantage is that each mode can be designed independently, but the disadvantage is that the user must constantly switch mode to perform the tasks he wants. The disadvantage may however be reduced if we can compose more complex modes from simpler ones. If done manually, this can help us make more of the tool functionality available in fewer modes. If done automatically, this could let us assemble task-specific modes or let the user select and activate several modes together, i.e. multi-selection of modes.

The mode identified by arrow in the toolbar in Figure 113 is in fact a quite complex mode, and can be considered a composition of a set of sub-modes for object selection, translation, rotation and resizing. In Figure 114, simplified models of the sub-modes are shown, following the pattern of Figure 110. Note that these models reflect how these would be designed without consideration of their composition. The SELECT mode detects mouse clicks on objects and represents the current selection visually for the user. The MOVE and RESIZE modes let the user drag objects and performs the actual move and resize operations, respectively.

A model of the composite SELECT, MOVE & RESIZE mode is shown bottom in Figure 114. The user may press the mouse button over an object and drag it to a new position. After dragging or just clicking on it, the object will be selected and resize handles will be shown. The handles are used for resizing the object. This model can be considered the result of a structured process of composition, where several techniques are used for making the resulting mode usable.



**Figure 114.** Composition of select, move and resize modes

The MOVE and SELECT modes have been integrated, by merging the corresponding states and transitions. The actions that are executed after the BUTTON RELEASED event triggers the final transition, have been combined. This direct way of integration is possible since there is no inherent conflict between moving and selecting an object. I.e. neither the feedback nor resulting actions conflict. The RESIZE mode cannot be directly combined in the same manner without conflicting with the MOVE mode's functionality, so it has to be separated in some way. The chosen technique is that of introducing visual handles, and adding conditions for triggering different transitions into the respective INSOURCE states. The top path of transitions correspond to the move case, while the bottom path is the resize path.

In the first integration step, the modes could be completely combined, while the second step resulted in completely separate paths. Something in-between is also possible, since the split is introduced to handle different cursor feedback for the two cases. If the INSOURCE feedbacks were the same, we could have moved the splitting of paths past the INSOURCE state. It is also possible to combine the paths later, if the subsequent actions are the same, but this is seldom the case since distinguishing feedback should persist until the gesture act is committed to.

## 6.7 Conclusion

We have shown how the RML and DiaMODL languages may be used for modelling concrete interaction. Two interaction styles have been modelled, the traditional forms-based style and direct manipulation. In both cases, we have shown how to link concrete interaction to abstract dialogue, to make the transition smooth.



## 6.7.1 Concrete interaction and the representation framework

RML is designed for modelling the static part of a domain, and has in this chapter been used for the domain of concrete interaction elements. DiaMODL targets the dynamic part of the user interface software, and covers both abstract and more concrete interaction elements. In terms of the classification framework introduced in Chapter 3, RML and DiaMODL have been used to cover the right part of the solution-oriented perspective, by focusing on the software solution to a design problem. The models have been hierarchically structured, and have covered a wide range of the granularity dimension. Both rough and detailed models of the concrete interaction elements have been made, i.e. the models have formalised aspects of concrete interaction of varying degree.

Below we will discuss how our usage of RML and DiaMODL in the context of concrete interaction supports the 6 movements in the design representation space introduced in Section 3.6:

- Movements 1 and 2, along the perspective axis: From problem to solution and from solution to problem, respectively.
- Movements 3 and 4, along the granularity axis: From down the hierarchy from high-level descriptions to lower-level ones, and up the hierarchy from lower-level descriptions ones to higher -level ones, respectively.
- Movements 5 and 6, along the formality axis: From informal descriptions to more formal ones, and from formal descriptions to less formal ones, respectively.

The relevant movements within the representation space is between concrete interaction and abstract dialogue models, up and down the hierarchy of interaction elements, and between formal and informal representations of interaction.

### Movements 1 and 2

DiaMODL has been used for modelling most aspects of concrete interaction, in particular the external interface of the standard concrete interaction objects. Hence, there is no conceptual gap to bridge in the transition from abstract dialogue to concrete, as was the case for the transition from task to dialogue. The transition involves matching the requirements of the abstract dialogue/interactors, i.e. gate interface and internal state space, with that provided by concrete interaction elements. The concrete interaction objects may require parameters that must be provided by the context it is inserted into, like containing windows and triggering conditions. Usually there will be several different concrete candidates, and the specific requirements of each may be used to narrow the choice. For instance, a candidate may only be allowed in a certain context, or may require to much window space.

Many standard concrete interaction objects have been given an interpretation in terms of the DiaMODL language. An abstract view of a concrete design e.g. made using a GUI builder, may be derived by using the DiaMODL interpretation of each element, and abstracting away look & feel-oriented details, like window containment structure and layout.

### Movements 3 and 4

We have used RML to model the hierarchy of concrete interaction objects that are found in standard window-based toolkits. The model includes constraints for which interaction objects that may be contained in others, and hence provides guidelines for how to decompose a concrete element. For instance, a FRAME may include a MENUBAR and several TOOLBARS and PANES. Models of more specific dialogue window types may be included and additional constraints be added, to comply with explicit design guidelines.

The constraints expressed in the RML model of concrete interaction objects may also be used for introducing containers, i.e. adding levels up in the hierarchy. For instance, a PANE with a MENUBAR must necessarily be contained in a FRAME.

### Movements 5 and 6

Informal representations are typically used the creative phases of design and when communicating with the end-user. This implies using concrete representations, i.e. representations that are aligned in perspective dimension with the ones discussed in this chapter. Hence, the translation from informal sketches to formal models of concrete interaction requires identifying the concrete interaction objects, and using the corresponding model formulated in DiaMODL. The opposite movement, from formal to informal, involves visualising the concrete interaction objects that are referred to in the formal model, e.g. building a GUI populated with objects that are instantiated from a specific toolkit.

The support of the six movements is in general fairly good. Additional support for movements 3 and 4 may be given part by strengthening the RML model of the concrete interaction objects and the constraints for how they are composed.

# Chapter 7

## Case study

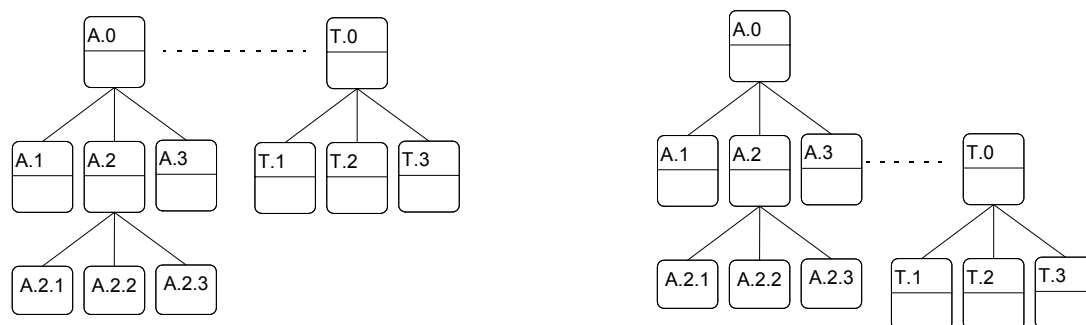
This chapter presents a complete example of our approach to model-based user interface design, using the modelling languages that have been presented in Chapter 4, “Task modelling” and in Chapter 5, “Dialogue modelling”. We illustrate the use of the task and dialogue modelling languages and the design approach by employing them on a case.

### 7.1 The IFIP case

In this chapter we illustrate the use of our task and dialogue modelling languages and approach to design, by employing them on a case, complementing the more fragmentary presentation of the languages in the previous chapters. We have chosen to use the IFIP working conference case, a case which previously has been used for comparing and evaluating IS design methods, including the APM workflow language which our task modelling language is partly based on. The case concerns how a conference is arranged, and in particular how the review process is coordinated. The problem definition, as presented in Figure 7.1, p. 226 in [Carlsen, 1997], is shown in Figure 116 on page 149. This (formulation of the) problem definition is not particularly suited for validating a user interface design method, as it does not describe the problem of arranging a conference from a user’s or user role’s perspective. To be a realistic case of a model-based and user-centred development approach, we would need access to real users, performing realistic tasks in a real environment. However, by taking the APM case study from [Carlsen, 1997] as a starting point and interpreting it based on our own experience with the reviewing process, we are able to illustrate our modelling approach and languages, and use it for discussing task and dialogue modelling in general and its relation to workflow in particular.

In Figure 12 on page 29, we suggest that workflow and task models essentially view a problem with the same problem-oriented perspective, but at different levels: Workflow is concerned with how actors within an organisation cooperate in reaching business goals, while task models focus on individual actors or small groups, and the tasks they perform to pursue their individual goals. This was part of the motivation for basing our TaskMODL language on APM, as described in Chapter 4, “Task modelling”. In line with this reasoning, we take Carlsen’s top-level workflow model, shown in Figure 117, as a starting point for our task model, and choose to focus on the *program chair role*. This prompts the question: At what level are the workflow and task models aligned? Figure 115 illustrates two alternatives: In the left part the top-level task T.0 is aligned with the top-level process A.0, while in the right

part the top-level task T.0 is aligned with the 1. level processes A.1, A.2 and A.3. In both cases the task hierarchy will be a projection of the workflow model with respect to the chosen focus on a particular role, in the sense that tasks performed by other roles are left out. The choice depends both on the scope of the workflow vs. the task model, and to what extent the workflow decomposition is allowed to constrain the task structure. The higher we “lift” the task model, the greater freedom is given to redesign the work structure according to the needs of the target user group, in our case the program chair. Biasing or balancing the needs of different user groups is complicated and often a political issue, and will not be further discussed here. It suffices to say that we have chosen the left alternative, i.e. our 1. level tasks are at the same level as the ones shown in Figure 117.



**Figure 115.** Workflow process and task hierarchy alignment

- *Left:* 0. level process aligns with 0. level task.
- *Right:* 1. level process aligns with 0. level task.

## 7.2 The ‘Organize’ task

For the purpose of this case study, i.e. to illustrate how our user interface modelling languages are used together, we have chosen to focus on the program chair role. The case study covers a subset of the tasks from the case: how the chair records responses to the Call for Paper and invitations, chooses the reviewers and collect their reports. These tasks are based on the processes of the overall workflow, as shown in Figure 117, and represent a subset of the processes the CHAIR role performs or participates in. Two alternative task models are shown in Figure 118 and Figure 119, where the numbering corresponds to the numbering in Figure 117. In both cases, the tasks correspond to processes in Figure 117 of the same name, but represent a reinterpretation according to the difference in focus: the goals and tasks of the CHAIR role.

In the model in Figure 118, the ORGANIZE task is decomposed into a *sequence* of subtasks (A.2-A.5) each taking a set of elements as input, processing them and delivering another set as output, as a trigger for the next task. The initial trigger is the responses to the Call for Paper and invitations, while the final result is a set of review summaries. As shown, the performing actor role is the CHAIR, while a set of REVIEWER actors is provided as contextual information. The responses are recorded in task A.2 and each of the papers are allocated to a set of reviewers in task A.3. The relation between a paper and a reviewer is represented by a review report, and the set of preliminary reports triggers both the reviewer’s performance of the review (task A.4) and the task (A.5) for collecting the final ones. Task A.5 should poll the parallel task A.4 and ensure that the reviews are performed before the deadline. A

### Problem Definition

#### 1. Background

An IFIP Working Conference is an international conference intended to bring together experts from all IFIP countries to discuss some technical topic of specific interest to one or more IFIP working groups. The usual procedure, and that to be considered for the present purposes, is an invited conference which is not open to everyone. For such conferences it is something of a problem to ensure that members of the involved IFIP Working Group(s) and Technical Committee(s) are invited even if they do not come. Furthermore, it is important to ensure that sufficient people attend the conference so that the financial break-even point is reached without exceeding the maximum dictated by the facilities available.

IFIP Policy on Working Conferences suggest the appointment of a Program Committee to deal with the technical content of the conference and an Organising Committee to handle financial matters, local arrangements and invitations and/or publicity. These committees clearly need to work together closely and have a need for common information and to keep their recorded information consistent and up to date.

#### 2. Information system to be designed

The information system which is to be designed is that necessary to support the activities of both a Program Committee and an Organising Committee involved in arranging an IFIP Working Conference. The involvement of the two committees is seen as analogous to two organisational entities within a corporate structure using some common information.

The following activities of the committees should be supported.

##### Program Committee:

- Prepare a list to whom the call for papers is to be sent.
- Registering the letters of intent received in response to the call.
- Registering the contributed papers on receipt.
- Distributing the papers among those undertaking the refereeing.
- Collecting the referees' reports and selecting the papers for inclusion in the program.
- Grouping selected papers into sessions for presentation and selecting chairman for each session.

##### Organising Committee:

- Preparing a list of people to invite to the conference.
- Issuing priority invitations to National Representatives, Working Group members and members of associated working groups.
- Ensuring all authors of each selected paper receive an invitation.
- Ensuring all authors of rejected papers receive an invitation.
- Avoiding sending duplicate invitations to any individual.
- Registering acceptance of invitation.
- Generating a final list of attendees.

#### 3. Boundaries of system

It should be noted that budgeting and financial aspects of the Organising Committee's work, meeting plans of both committees, hotel accommodation for attendees and the matter of preparing camera ready copy of the proceedings have been omitted from this exercise, although submissions may include some or all of these extra aspects if the authors feel so motivated.

**Figure 116.** IFIP conference arrangement problem definition

review summary for each paper is made from the set of reports for the paper, and the set of summaries is the final result of the top-level ORGANIZE task. This would be the input to a task A.6 for deciding which papers should be accepted or rejected.

Characteristic for the first ORGANIZE model is the batch-like processing of sets, each step taking a set as input and outputting another set as a trigger for the following task. It may not be realistic to finish one step completely before initiating the next, for two reasons:



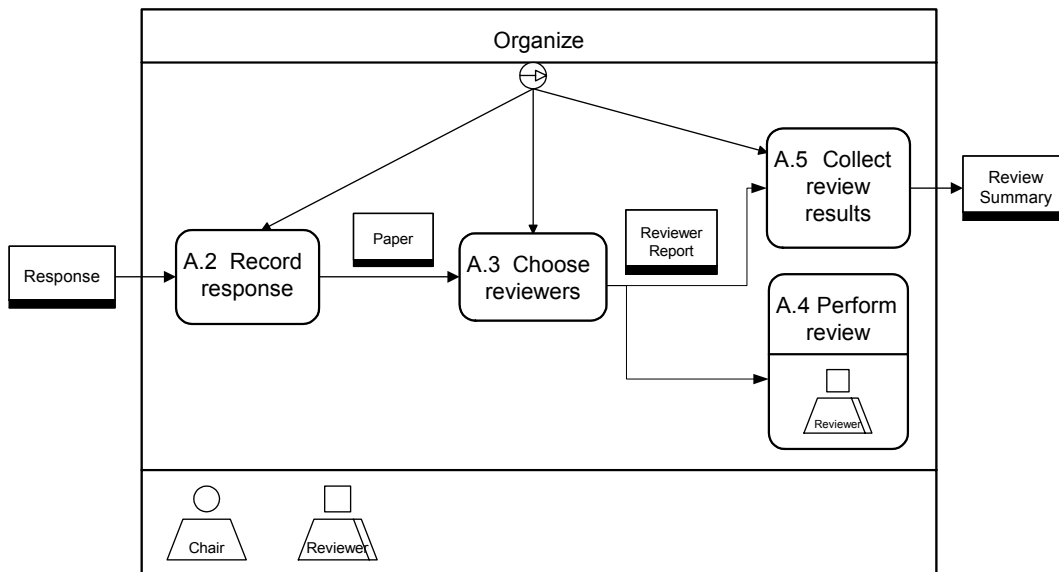


Figure 118. Sequence of set-processing tasks

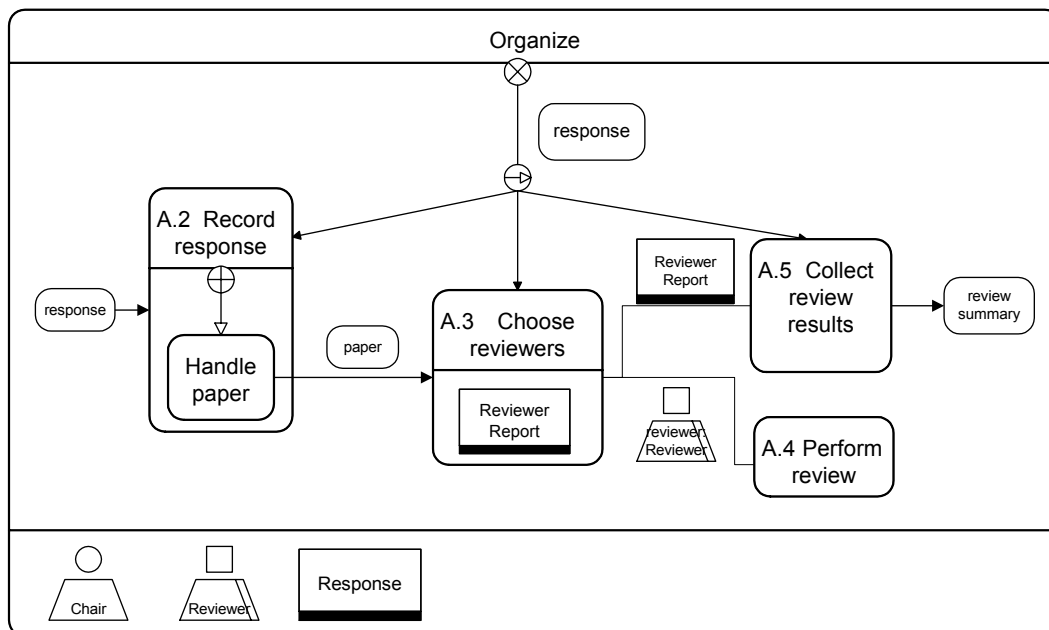


Figure 119. Aggregation of individual task sequences

- Although a step will for most practical purposes end before the next starts, there may be exceptions, like allowing late-comers or misplaced papers to be processed.

In the alternative model, as shown in Figure 119, each task is triggered by a single element, and produces its result based on this element alone. While the task flow is still sequential, this constraint relates to the tasks for individual papers, giving a set of sequences for individual elements, instead of a single sequence for sets of element. In the model this is expressed by using the sequence relation (arrow) across the tasks for each paper and unconstrained aggregation across each paper sequence. The consequence is that different tasks for different papers may be active at once, resulting in a more complex task structure. This identified difference can have three effect on the design (process):

- The interface will become more complex, if it is designed to mirror the complexity of the task structure.
- The interface must be designed more carefully, to keep it simple while supporting the expressed task structure.
- The interface must be simplified by deliberately changing/forcing the task structure to that of the top model.

In the general case, there may be good reasons for each of these, depending on factors like the users' experience, dimensions of the target display, ergonomics of the target input device and the development budget. In any case, we note the importance of considering the consequences of different models of the "same" task domain.

In addition to allowing a more interleaved task performance, the second model has two other notable features. First, it expresses that there is one particular variant of task A.2 that produces a paper, and presumably others that do not. The full decomposition of A.2 should show when and why a paper is produced. Second, it shows that for each paper there is a set of review reports, i.e. the total set of review reports may be grouped by the paper it reports on. The second model may seem more detailed than the first, since it shifts the focus from the handling of sets to individual elements. For the first model, the details of how each element is handled can be introduced in the decomposition of each top-level task, in the manner shown in Figure 44, in Chapter 4.

### 7.3 The domain model

Both of the top-level models of Figure 118 and Figure 119 introduce concepts of the task domain, e.g. REVIEWER, PAPER and REVIEWER REPORT. These and others are usually described more explicitly in a conceptual model of the static domain. Some prefer to identify the concepts through the task model, while others prefer making the static domain model first. A conceptual model of the IFIP case is shown in Figure 120.

The main features of this model are:

- The definition of several actor roles, with potentially overlapping populations.
- The PAPER concept, as a specialisation of RESPONSE (to a REQUEST), which contains CONTENT.
- The REVIEWER REPORT concept relating a REVIEWER to a PAPER.
- The REVIEW SUMMARY as an aggregation of a set of REVIEWER REPORTS.
- The CONTENT concept, which captures that there are different "physical" representations or formats of a paper, including the manual paper format and two file types.

Together the task and domain models represent important knowledge about the users' universe of discourse, that must be supported by the user interface.



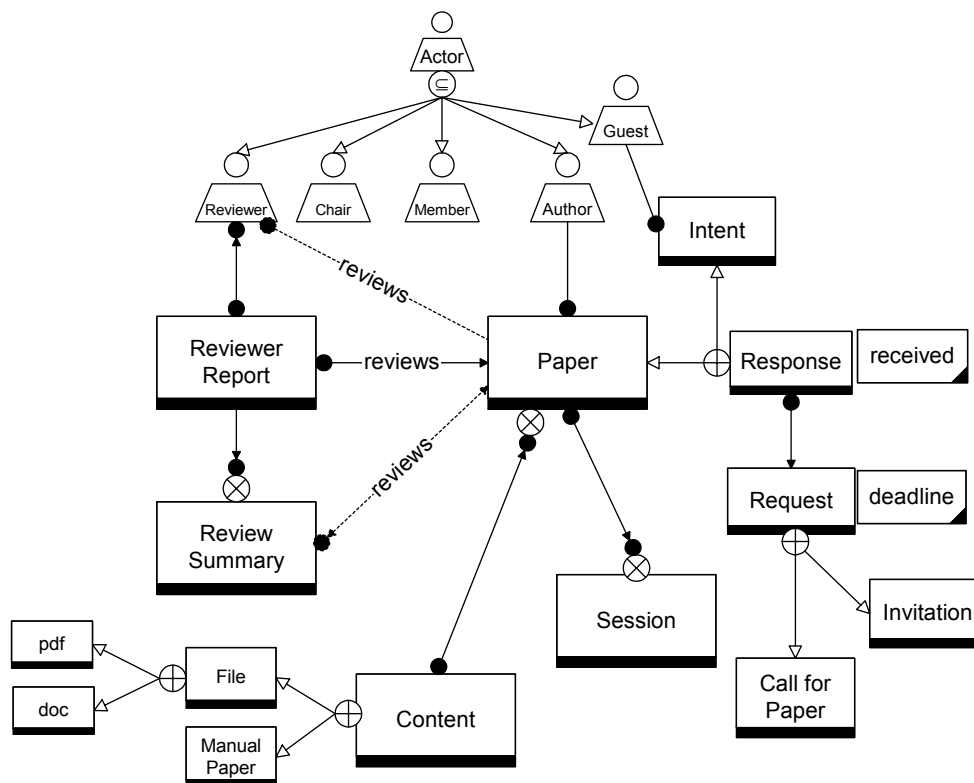


Figure 120. Conceptual model of the static domain

## 7.4 The main subtasks

The level one subtasks will be decomposed in the following sections.

### 7.4.1 Task A.2, Record response

Task A.2, Record response, has the goal of registering a response and ensuring there is a PDF version of the paper. Its decomposition is shown in Figure 121. Note the following:

- The RESPONSE element that is the input to A.2, is the same element as the one output by the HANDLE PAPER task, as the name indicates. Note however, that the input element may refer to an instance that is not yet represented in a system, e.g. physical letter or email. The output element, on the other hand, will presumably be explicitly represented in the information system.
- The model specifies that several decisions must be made: Is the RESPONSE too late, is it a letter of INTENT and is it a MANUAL PAPER that must be scanned? It should not be too difficult to design an email or web interface, so that RESPONSES could be handled automatically, but this may not be a relevant goal. The current model does not commit to any particular way of making the decisions.

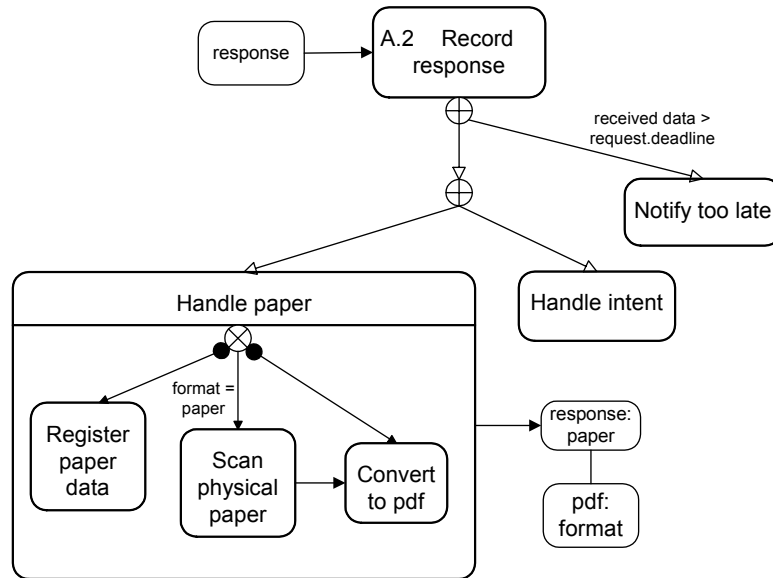


Figure 121. Task A.2: Record response

- Registering paper data and converting to pdf are required subtasks, while the scanning task is conditional/optional.
- The task model does not require registering the paper data before converting the paper to pdf, which in turn means that the dialogue implementation must be able to handle “dangling” files that are yet unrelated to a response.

### 7.4.2 Task A.3, Choose reviewers

In task A.3, the goal is to assign a set of REVIEWERS to a PAPER, by creating a set of REVIEW

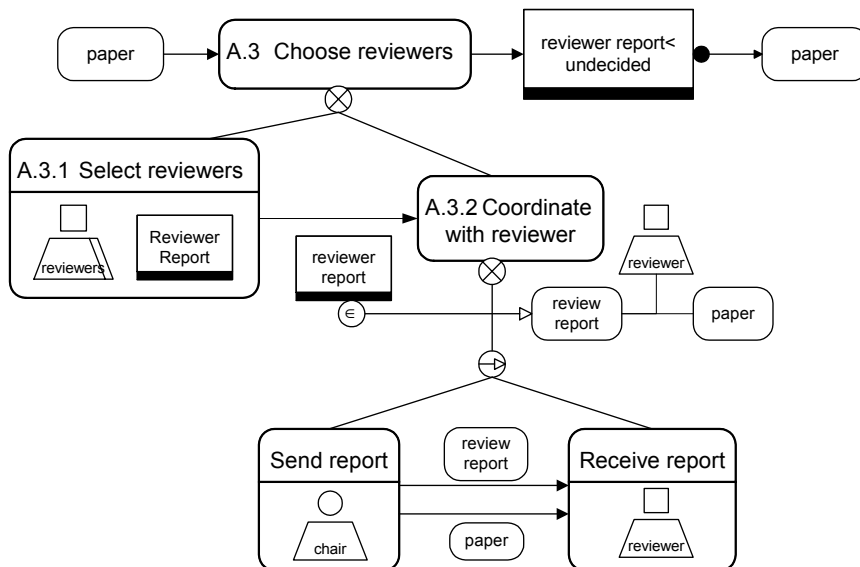


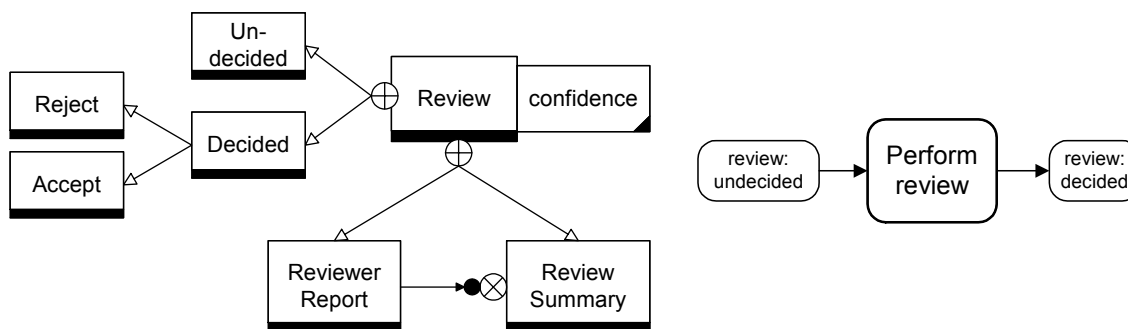
Figure 122. Task A.3: Choosing reviewers for a paper and sending the review report

REPORTS for the PAPER input element. Each REVIEW REPORT will initially be classified as UNDECIDED. We have in addition included the sending of the PAPER and REVIEW REPORT to the REVIEWER. A model of the task is shown in Figure 122. The first subtask (A.3.1) selects

the reviewers, based on the potential REVIEWERS and existing assignments represented by the extension of the REVIEWER REPORT concept, e.g. to prevent assigning too many papers to a single reviewer. The result is a set of newly created REVIEW REPORTS, which is used as input to the other subtask (A.3.2). Then each PAPER and REVIEW REPORT is sent to the corresponding REVIEWER. Note that the task of receiving is included, and that the performing actor of this task is indeed the REVIEWER related to the REVIEW REPORT. As shown, we have chosen to send each reviewer individual review reports. The alternative of sending them as a set would perhaps have been better for the reviewer, and more natural if we had chosen to decompose the set-oriented pipelined model in Figure 118.

### 7.4.3 Task A.4, Perform review

The goal of the review process is to classify a paper as either rejected or accepted, but this distinction is not expressed in the static model in Figure 120. We first need to introduce the different classifications, and define them as specialisations of REVIEW REPORT. Since a REVIEW SUMMARY may belong to the same classes, we also introduce a REVIEW generalisation, which is specialised in independent ways, as shown left in Figure 123. In the right model fragment we have expressed that the goal of the REVIEW PAPER task is to change a REVIEW from the UNDECIDED state to the DECIDED state.



**Figure 123.** Refining the Review concept  
 • *Left:* Refined concept model.  
 • *Right:* Explicit pre- and post-conditions of task.

### 7.4.4 Task A.5, Collect review results

For the CHAIR actor, the main task related to review is collecting the REVIEW REPORTS for each PAPER and making a REVIEW SUMMARY. The REVIEW SUMMARY is related to the corresponding REVIEW REPORTS through an aggregation relation, and in addition it represents a decision on its own, i.e. is classified as DECIDED (REJECT or ACCEPT) or UNDECIDED, according to the policy of conference. The model in Figure 124 shows how this task may be decomposed. As shown, the input is a set of REVIEW REPORTS and the output is a REVIEW SUMMARY. After summarizing the reports one of three different subtasks is performed. The case of an insufficient number of reports is give special treatment. If there is a clear majority for REJECT or ACCEPT, the REPORT SUMMARY is classified accordingly, otherwise the CHAIR must decide himself.

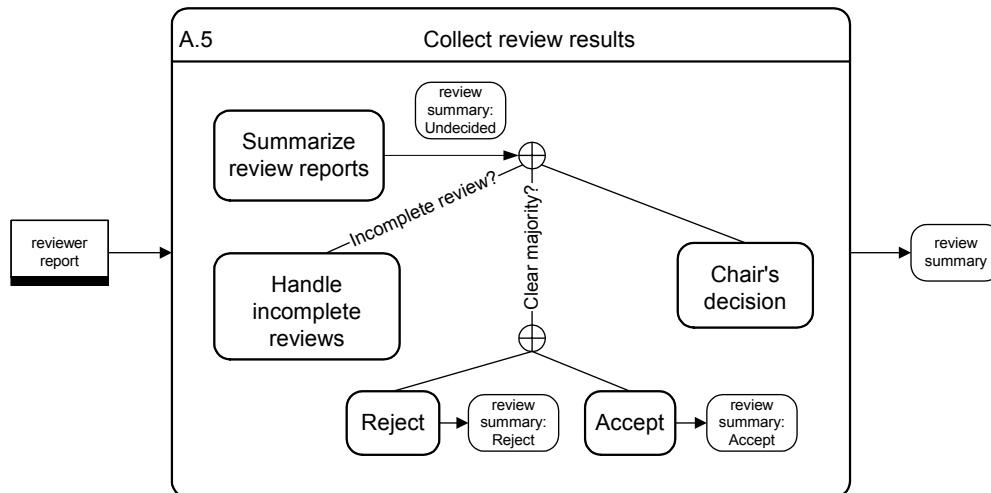


Figure 124. Task A.5: Collecting the review results and making a review summary

## 7.5 Dialogue-oriented domain model - considerations

The conceptual model shown in Figure 120, is supposed to capture the users' view of the static domain. The task models presented in the previous sections, describe how the users' work may be structured. Both models may be considered as strong requirements that the user interface dialogue must fulfill. Alternatively, the two models may be taken as a starting point from which a dialogue is designed, and in the process two dialogue-oriented conceptual and task models are derived. For instance, we may choose to remove constraints expressed in the domain model to make the application more flexible, or simplify or constrain the task structure to make the interface easier to learn. In this case study we will limit ourselves to pointing out some aspects of the domain model that should be considered when designing the dialogue:

- The REVIEWER actor is defined as an actor that is necessarily related to a REVIEW. This seems like a reasonable intentional definition, but may in fact be a problem in the context of the dialogue design. The reason is that the model defines a REVIEWER as someone who *has already* reviewed a paper, not someone who *can*. According to the REVIEWER definition, an actor currently not assigned to a paper will not become a REVIEWER until after A.3 has been performed. Hence, the interface must allow selection among a set of *potential* REVIEWERS and not the *current* ones, or the REVIEWER definition should be changed by including 0 (zero) in the cardinality of the REVIEWS relation.
- Several concepts require a relation to at least one element of another concept. For instance, a REVIEW SUMMARY must contain at least one REVIEW. Note however, that a computer system may need to allow 0 (zero) such relations, since it may be natural for the user *first* to create an empty SUMMARY and *then* to add the REVIEWS, as they are provided by the A.5 task.

- Along similar reasoning, the RESPONSE elements need not in practice have a relation to a REQUEST element, since these are not used in the task. It may be the case that INTENT responses need to be matched against invitations, but the CALL FOR PAPER will not necessarily be represented explicitly in the user interface, although it is part of the conceptual model.

These comments are related to the difference between focusing on the problem perspective, i.e. task and domain, and the solution perspective, i.e. dialogue and concrete interaction. The user may have a different understanding of the domain, than what is reflected in the design. For instance, the PalmOS and Pocket PC have different ways of structuring data, using categories and hierarchies, respectively and although the user may be used to hierarchies, a Palm application may stick to using categories. In addition, it may be desirable to design a user interface with greater flexibility than what is required by the domain and task models. Empty containers is one example; they may be useful as an intermediate state in a dialogue, but rather pointless in a conceptual model. In sum, this means that the conceptual model should be reconsidered and revised before using it as a basis for dialogue design and implementation.

## 7.6 Designing a dialogue for each main subtask

The tasks modelled above, are to be supported by a set of dialogues, which are *composed* into a user interface. In terms of the representation space introduced in Section 3.3 and illustrated in Figure 11 on page 29, this corresponds to a movement along the problem-solution axis, from problem to abstract interaction. One way of designing the user interface is to design each dialogue separately, and then merge the dialogues using operators for composition. To avoid duplicating functionality, similar dialogues that are active together may be combined into one. Powerful interaction objects provided by the underlying toolkit, and modelled as suggested in Chapter 7 (e.g. Figure 104 on page 132), may further be utilised for reducing the number of separate dialogue elements.

We present the user interface design as a top-down process, moving from task via abstract dialogue to concrete dialogue. This is not the only way of conceiving a design, and should not be considered a suggestion for “a best way” of designing. As discussed in Chapter 3, “Design and representation”, it may be just as desirable to design using more solution-oriented and concrete representations like sketches, low-fidelity prototypes and mock-ups, to stimulate creativity and provide means of communicating with end-users. The abstract and formal models may then be built, as part of the process of clarifying and making the design more explicit. Alternative concrete design suggestions may further be made by moving from abstract to concrete dialogue, in the manner presented below in Section 7.8.

## 7.6.1 Registering a paper

Task A.2, RECORD RESPONSE has the goal of making a new response element and in the case of a paper response, preparing an electronic version of the paper content. We limit our dialogue model to the handling papers, and split it into four interactors and a function, as shown in Figure 125:

- REGISTER PAPER DATA supports entering data about a paper, e.g. the author(s), received date, keywords etc. The result of interaction is a PAPER element.
- CONVERT TO PDF supports converting the paper content to the pdf format, e.g. by scanning in a physical paper or running a file through a pdf filter. The output is PDF element.
- The X (for aggregation) function receives the PAPER and CONTENT and creates an aggregation relation. The output is the same PAPER, which is presented to the user by means of the third interactor.
- The PAPERS interactor is used for presenting the set of PAPERS and highlighting one PAPER element in the set. This interactor is a variant of the one shown in Figure 62 on page 93. The addition of an output/receive gate for the element, makes it possible to control which element is highlighted from outside the interactor. Hence, the selected PAPER element may be controlled by the user, and hence be received as input on the input/send gate, and by a connection attached to the output/receive gate. The content of the selected element, whether controlled by the user or attached connection, is presented to the user by means of the fourth interactor.
- The CONTENT interactor presents the content part of a paper, received through the output/receive gate.

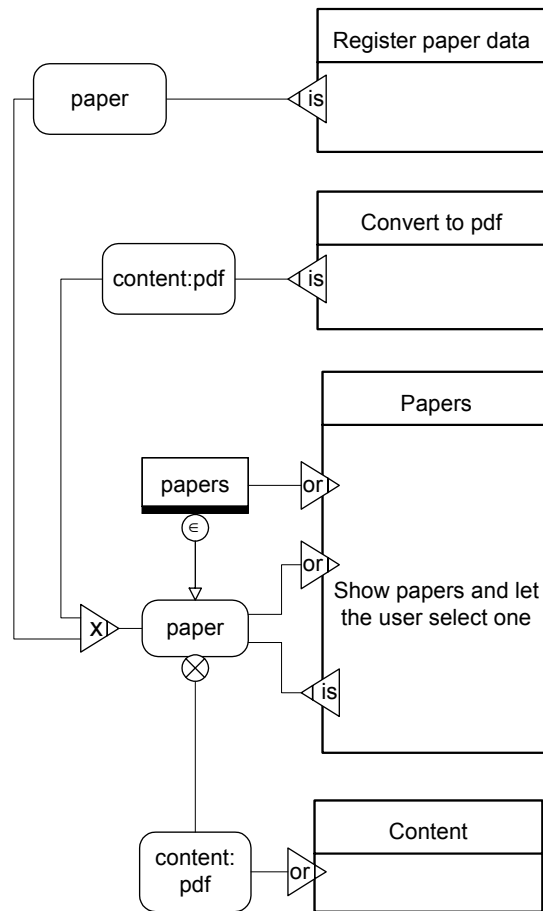


Figure 125. Registering a paper

The first two interactors can be argued for based on the task model, since they represent tasks that the user interface must support. The PAPER and CONTENT interactors are not necessary in the sense that they are motivated from the task model, but they provide valuable feedback from the result of interacting with the two former interactors. For instance, if the task of registering a set of received papers is interrupted, the displayed set of papers can be examined to find which paper to resume with.

All these interactors are initially planned to be active simultaneously, which in Statecharts terms corresponds to and-composition. This might change however, when an interaction style and look & feel is chosen. For instance, the process of converting a file to pdf may be implemented by a *modal* dialogue, due to Adobe's implementation of the standard filtering software. The display of the pdf content may similarly be difficult to compose into the same window as the other interactors, given the size needed by the standard Acrobat Reader component. Hence, the selection of concrete interaction objects may greatly influence the dialogue design.

## 7.6.2 Assigning a set of reviewers to a paper

According to the model of task A.3 in Figure 122, the input to the task of choosing reviewers is a PAPER, while the set of (potential) REVIEWERS and current set of REVIEW REPORTS are provided as contextual information. This means that an interactor for performing this task, should present the corresponding domain data, and this is the basis for the interactor models shown in Figure 126 and Figure 127.

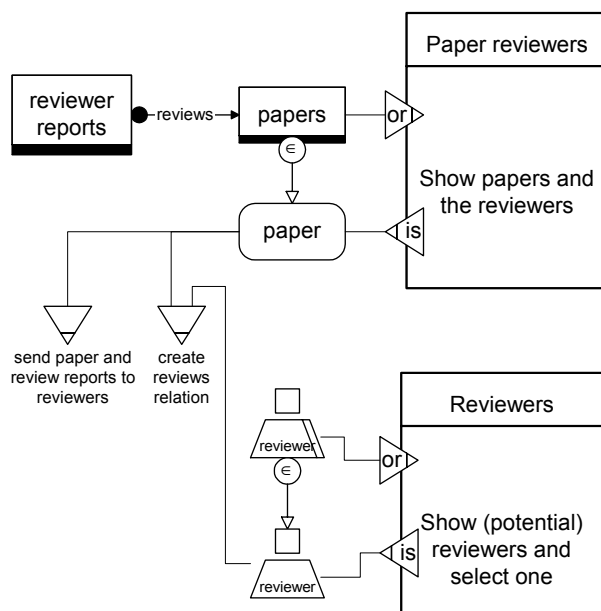


Figure 126. Assigning reviewers to papers.

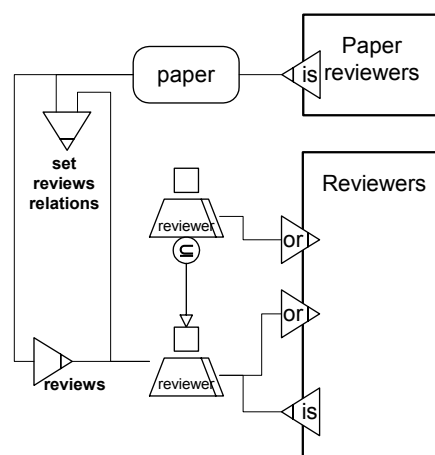


Figure 127. Alternative design for assigning reviewers to papers.

The PAPER REVIEWERS interactor, receives and outputs to the user a set of PAPERS, while letting the user select a PAPER. We have included the REVIEWER REPORTS set and REVIEWS relation in the model fragment, to highlight the part of the domain model that the user is interested in experiencing. The REVIEWERS interactor receives and outputs the set of (potential) REVIEWERS, and provides the user means for inputting a single REVIEWER. The input/send gates for the selected PAPER and REVIEWER are connected to a function for creating a REVIEW REPORT related to the selection. Another function takes the selected PAPER as input and sends the PAPER and related REVIEW REPORTS to the REVIEWERS.<sup>1</sup>

1. To avoid sending the PAPER twice (to the same REVIEWER), the domain model should perhaps be revised to include a REVIEW REPORT attribute indicating whether it has been sent or not.

A problem of this design is that the REVIEW REPORTS must be made one by one. In addition, the REVIEWERS interactor that is used for selecting reviewers does not indicate which reviewer that already has been chosen. An alternative design targeting these problems is shown in Figure 127.<sup>1</sup> This alternative design is based on a more direct coupling between the two PAPER REVIEWERS and REVIEWERS interactors and the domain data that these provide access to. The main difference is in the output/receive and input/send gates of the REVIEWERS interactor. This interactor is now specified as capable of both outputting and inputting a *subset* of the REVIEWERS, as well as outputting the whole REVIEWERS set. When the REVIEWERS subset is controlled by the output/receive gate, the subset that the user sees is the set of REVIEWERS related to the selected PAPER through the REVIEWS derived relation. Hence, when the user selects a different paper using the PAPER REVIEWERS interactor, the corresponding REVIEWERS subset presented by the REVIEWERS interactor will be updated. When the REVIEWERS subset is instead controlled/changed by the user and reflected in the input/send gate, it may be used for explicitly setting the set of review reports that exist and are related to the selected PAPER. This is achieved by replacing the function for creating a single REVIEW REPORT for a PAPER and REVIEWER pair, by a function for managing the set of REVIEW REPORTS for a certain PAPER. This function must be able to both create and delete review reports depending on how the user manipulates the value in the input/send gate of the REVIEWERS interactor.

### 7.6.3 Collecting and summarizing review results

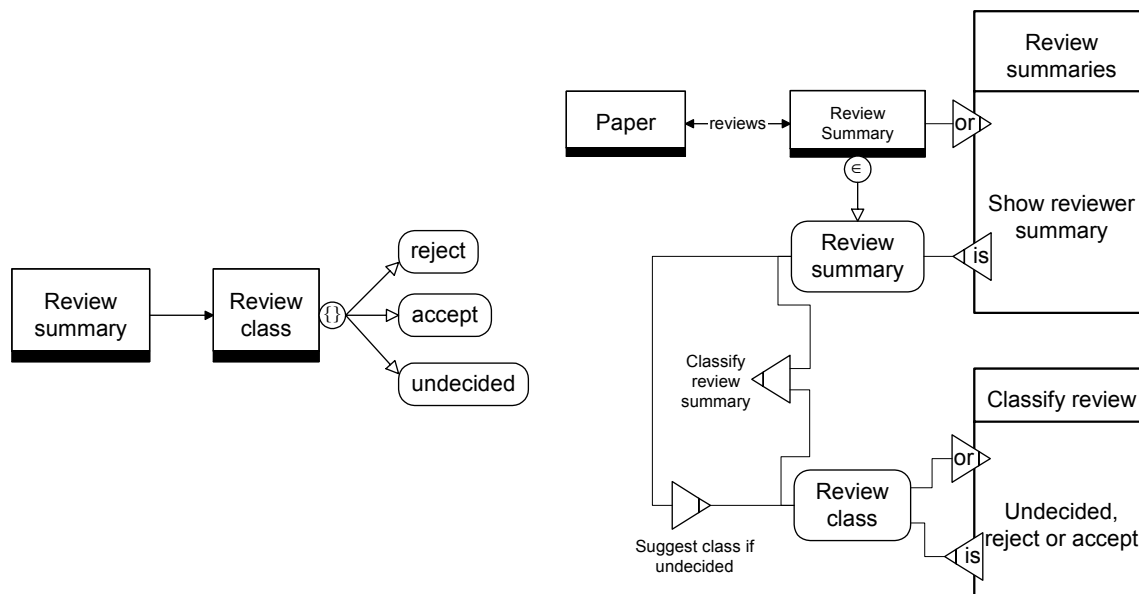
The final task that we are providing dialogue support for is concerned with classifying review summaries. Recall from the domain model in Figure 120 that there is a one-to-one relation between a PAPER and a REVIEW SUMMARY, as indicated by the bi-directional arrow. Our design assumes that the review summary is automatically created when the review reports start ticking in, so that classification reduces to selecting a review summary (or a paper) and a review class and invoking a function. However, first we need to introduce the concept of a REVIEW CLASS, i.e. an explicit concept for the different classifications of a REVIEW, and relate it to the REVIEW SUMMARY, as shown top left in Figure 128. Then, as shown right in the same figure, the user can select a REVIEW CLASS using the bottom CLASSIFY REVIEW interactor, and (re)classify the REVIEW SUMMARY that is selected using the top REVIEW SUMMARIES interactor.

In the task model three cases are distinguished, one for handling incomplete reviews, one for automatic classification based on majority, and one where the CHAIR decides. In our design, we want to help automate the classification process, while leaving the final decision to the user. When a REVIEW SUMMARY is selected, a REVIEW CLASS value is computed by a function (named “suggest class if undecided” in the figure), before being output by the CLASSIFY REVIEW interactor through the output/receive gate. The function is supposed to compute a suggested class if the current class is UNDECIDED, or else pass the current (DECIDED) class through.

---

1. For the PAPER REVIEWERS only the input/send gate is shown, this interactor is the same as the one in Figure 126.



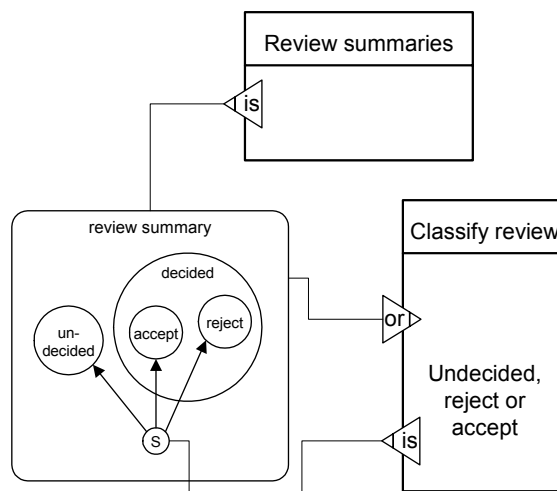


**Figure 128.** Classifying review summaries.

- *Left:* A review class concept is introduced to allow explicitly outputting and inputting the classification of a review.
- *Right:* Selecting a review summary and a review class and invoking a classification function.

A different view of the CLASSIFY REVIEW interactor is shown in Figure 129. This model relies on the duality between disjoint concept specialisation and or-decomposed states, as discussed in Section 4.6.3 and exemplified in Figure 27 on page 59. The model is complementary to the model in Figure 128, by more directly expressing how the user changes the class of the REVIEW SUMMARY element by means of the CLASSIFY REVIEW interactor.

Each REVIEW SUMMARY is modelled as a state hierarchy that corresponds to the classification hierarchy for the UNDECIDED, DECIDED, ACCEPT and REJECT classes. The (current state of) the selected REVIEW SUMMARY is attached to the output/receive gate of CLASSIFY REVIEW, and hence is presented to the user. The input/send gate is attached to and controls a Statecharts selection transition, to let the user change the current state and hence class of the selected REVIEW SUMMARY.



**Figure 129.** Classification as a state transition

## 7.7 Composing the dialogues - considerations

The dialogue models presented in Section 7.6, may be composed in many different ways, within the abstract dialogue perspective and modelling language. In terms of the representation space introduced in Section 3.3 and illustrated in Figure 11 on page 29, this corresponds to a movement along the granularity axis, from medium to higher level. In our Statecharts-based language, the basic composition operators are and- and or-composition, as discussed in Section 5.3.2. The choice between these two depends on both the task models in Section 7.2 and Section 7.4 and the target platform, interaction style and look & feel. In addition, if two or more interactors are similar, it may be possible to merge them, e.g. because they perform the same function and anyway are active at the same time, or because they have similar functions and there exists a concrete component support both at the same time.

As discussed in Section 7.2, the set-oriented version of the ORGANIZE task shown in Figure 118 is sequential and hence suggests using or-composition. The other variant shown in Figure 119 has no such constraint on the top level, although it is sequential for each PAPER. Since the subtask models have focused on the latter variant of the top-level model, it seems that and-composition is best suited for our dialogue. We have nevertheless chosen something in-between, based on the several observations:

- We want to make the most relevant tasks easiest to perform, and hence the sequential nature makes it natural to structure the dialogue into views that may be switched between
- All the dialogues have one interactor that provides access to PAPERS and related elements through the REVIEWS relations. Hence, it may be possible to combine them and and-compose this merged interactor with more task specific dialogues/interactor structures.
- The complexity of an and-composed dialogue may too high for the user and may not be well supported by a standard display.

The model shown in Figure 130, illustrates our composed design. At the top level, two main states are and-composed. The upper compartment contains an interactor for outputting a set of PAPERS and other elements (REVIEW SUMMARIES and REVIEW REPORTS) that the PAPERS are related to through the REVIEWS relation. The interactor also supports inputting or selecting one of the PAPERS. The lower compartment contains three or-composed (disjoint) sub-states labelled with the section numbers where their corresponding dialogues are presented. Each of the substates contains the same interactors as the previously presented dialogues, *excluding* the PAPER interactors. The respective PAPER interactors have instead been *merged* and placed into the top compartment. An interactor attached to a selection transition, has been added in the lower compartment. Note that this interactor is actually *and-composed* with both the upper and lower compartments, since it is not connected to other interactors or states with a *transition*, although it has a data connection. Using this interactor, the user can select among the substates, and hence one of three task-oriented sub-dialogues. For clarity, the model in Figure 130 omits the connections that provide the other sub-interactors with the selected PAPER element, and the functions that may be invoked. These should be added

to complete the model. In the case where a review summary is required instead of a paper, a REVIEWS function must be inserted or the connection labelled with the REVIEWS relation.

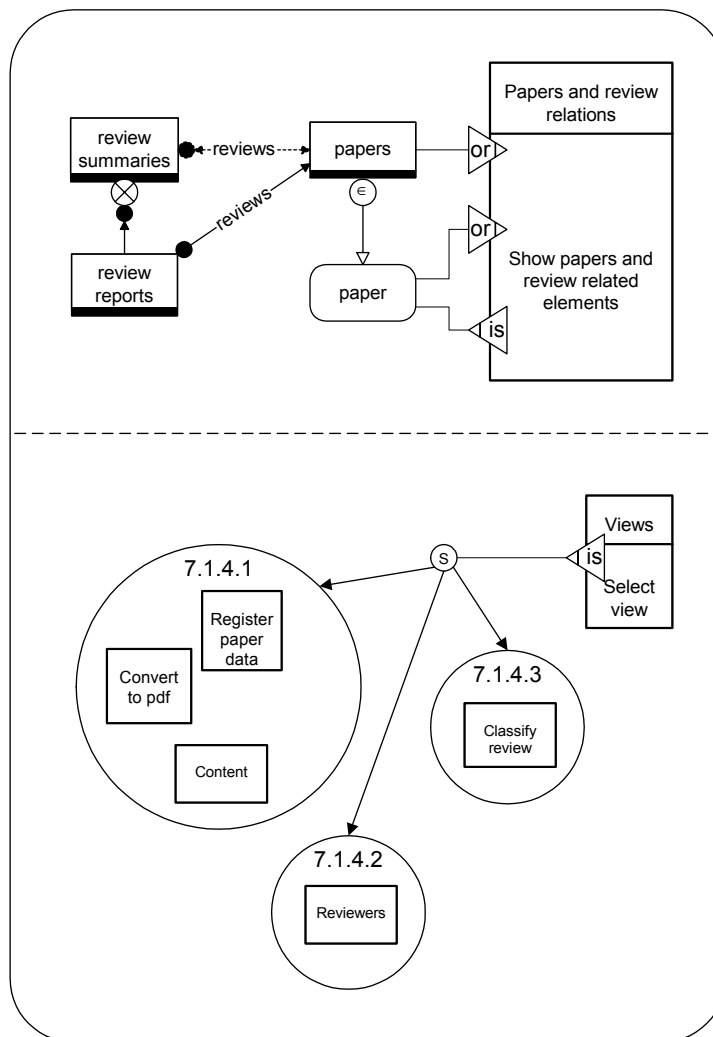


Figure 130. And-composition of common interactor with sets of task-oriented interactors

## 7.8 From abstract dialogue to concrete interaction

Our dialogue design is so far expressed in an abstract dialogue language, and when moving to concrete design elements, the abstract design may be mapped to many different platforms and concrete interaction objects. In terms of the representation space introduced in Section 3.3 and illustrated in Figure 11 on page 29, this corresponds to a movement along the problem-solution axis, from abstract to concrete interaction. Although the design model until now abstracts away look & feel, it does require functional abilities that may rule out limited platforms, such as PDAs. For instance, the PAPER AND REVIEW RELATIONS interac-

tor is expected to require some screen space (if the target platform is screen based), and in addition must be active at the same time as either of the more task specific interaction objects.

We choose a frame- and pane-based look & feel, based on standard interaction objects from graphical desktop interfaces, like buttons, text fields, listboxes and menus. The and-composed main dialogue of Figure 130, nicely maps to non-overlapping panes, and the or-composed sub-dialogues map to totally overlapping panes (or a single shared pane with interchangeable content) that the user may switch between. Each part of the composed model must then be interpreted in terms of this look & feel. In terms of the representation space introduced in Section 3.3 and illustrated in Figure 11 on page 29, this corresponds to a movement along the granularity axis, from high to middle and low levels.

The PAPER AND REVIEW RELATIONS interactor of Figure 130, focuses on the set of PAPERS and shows the REVIEWS relation relative to these elements. A concrete interaction object that supports this is a two-level folder view, similar to the one used by Windows Explorer. The set of PAPERS will occupy the first level and the REVIEW REPORTS that are related through the REVIEWS relation will occupy the second level, as illustrated in Figure 131. For each PAPER the classification of the corresponding REVIEW SUMMARY is indicated, and the number of REVIEW REPORTS supporting this classification is shown. We anticipate that such derived information will be useful, but this should be validated in user tests. We could also try out alternative designs for each kind of element. For instance, based on the discussion in Section 6.3.1 we could use the background colour for indicating the REVIEW CLASS, say green for ACCEPT, yellow for UNDECIDED and red for REJECT for both PAPERS/REVIEW SUMMARIES and REVIEW REPORTS.

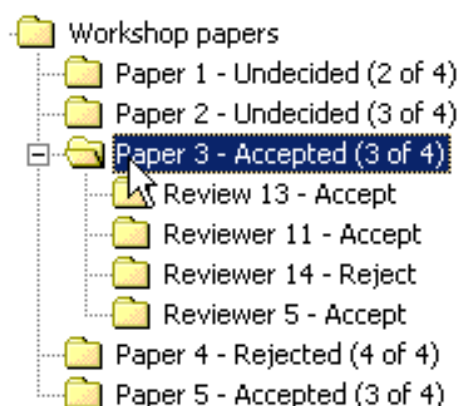
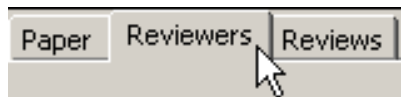


Figure 131. Folder view of papers and related reviews

This illustrates an important point when going from abstract to concrete interaction. When an appropriate concrete interaction object within the desired look & feel is chosen, we should analyse the expectations the users have to this particular interaction object and the capabilities provided by it. For instance, the folder view usually supports an element dependent icon and foreground and background colouring, in addition to the text. The user may also expect to have a set of actions available in a contextual menu, which may be popped up by clicking on an element with a special combination of mouse button and keyboard modifiers.<sup>1</sup> In our case, the task of classifying a PAPER/REVIEW SUMMARY may be allocated to the contextual menu, i.e. the contextual menu would implement the CLASSIFY REVIEW interactor. The act of clicking on a PAPER/REVIEW SUMMARY and selecting a REVIEW CLASS in the popup menu, provides the necessary information for the CLASSIFY REVIEW summary function included in Figure 128. To implement suggesting a default review class, i.e. setting the value of the output/receive gate of the CLASSIFY REVIEW interactor, we could position the popup menu so that the suggested class is selected by default.

1. In Windows the contextual menu usually is activated by the right mouse button, while on the one-button Mac a modifier is used.

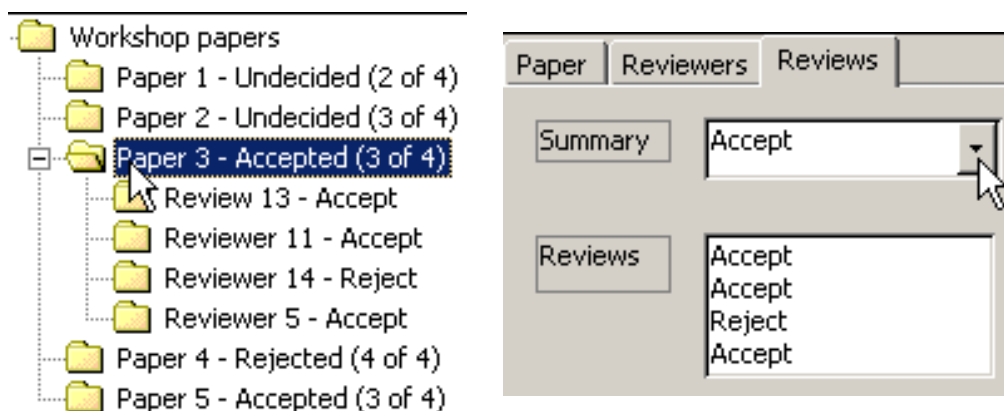


**Figure 132.** Tabs of tabbed pane

Each of the or-composed task-oriented interactors must be interpreted in terms of the same look & feel, as must a mechanism for switching between them. We will use a tabbed pane for switching, since it implements the desired or-composition behaviour. Three tabs are needed, e.g. labelled “Paper”, “Reviewers” and “Reviews”, as shown in Figure 132. An alternative design is a “View” menu with three checked menu items corresponding to the three tabs. However, since the tabbed pane has only this usage, while menu items are generic action objects, we prefer the former. A third possibility is using a single selection interaction object like dropdown listbox or group of radio-buttons. Such a design would correspond to viewing a dialogue as a piece of data, i.e. a kind of meta-model of dialogs. This view is more appropriate for a meta-task like tailoring/customisation so this third alternative is also discarded.

Designs for the three sub-dialogues of the tabbed pane are shown in Figure 133, Figure 134 and Figure 135. The “Paper” tab includes interaction objects for entering data about a PAPER, i.e. a standard fill-in form. The interaction objects must be chosen according to the underlying data type, in this case we have assumed all the three attributes have a text representation and hence have used text input fields. The “Reviewers” tab contains a listbox which must support highlighting (output) or selecting (input) one or more REVIEWERS. If the list is expected to be large and scrollable, we should instead use a design where the selection (subset of REVIEWERS) is shown in its own list. This case is similar to (excluding the filter) the dialogue for gas contract selection discussed below in Section 8.2, so one of the designs in Figure 139 may be used. The “Reviews” tab provides a dropdown listbox for selecting a REVIEW CLASS. Since the set of REVIEW CLASSES is small, we might as well have used radio-boxes, since they have the same interactor signature as the dropdown listbox, as discussed in Section 5.2.3. The list below the dropdown shows the related REVIEW REPORTS, as an aid in deciding which REVIEW CLASS to select. A means of looking at each REVIEW REPORT in detail should perhaps be added, either to this dialogue or the PAPER folder view.

The “Reviews” tab does not include any reference to the PAPER that the REVIEW SUMMARY and REVIEW REPORTS are related to. This information has been left out, since the tabbed pane is supposed to be laid out next to the folder interaction object where the corresponding PAPER already is highlighted, as shown in Figure 136. If the tabbed pane instead was part of a separate frame or model dialogue, an interaction object for indicating the PAPER should be included, e.g. by using the frame title.



**Figure 136.** Folder view of papers and review reports provides a context for “Reviews” tab

**Figure 133.** The “Paper” tab:  
Text input fields for the paper’s attributes

**Figure 134.** The “Reviewers” tab:  
Listbox supporting multiple  
selection of reviewers

**Figure 135.** The “Reviews” tab:  
Dropdown listbox for selecting a review  
class for a paper/review summary, and the  
corresponding list of review reports classes.

## 7.9 Completing the design

The structure of dialogues and interaction objects presented so far, provides the core user interface of an application for managing papers and reviews. To complete the application, we may extend the design in two directions.

The first direction concerns covering a larger part of the original process, as modelled in Figure 117, e.g. designing dialogues for supporting process A.7, “Grouping Accepted Papers into Sessions”. This design task will consequently require that we cover the SESSION concept and the aggregation relation to sets of PAPERS, as shown in the domain model in Figure 120. Incrementally adding functionality to an existing design indicates how well the design scales, and may require us to reconsider the whole dialogue structure.

In the case of process A.7 and the SESSION concept, we need to be able to present the SESSION aggregation - PAPER part of the domain model. We will utilise the flexibility of the folder view interaction object. The folder view shown in Figure 131 may be extended to

show this relation, by adding a session level in the hierarchy. I.e. the top level folder would contain SESSIONS, the next level PAPERS/REVIEW SUMMARIES and the third level the REVIEW REPORTS. The task of grouping PAPERS into SESSIONS could be supported through drag & drop, which is an interaction technique that the folder view interaction object usually implements and gives the user expectations of be able to use. A dummy SESSION would be needed to initially contain the PAPERS that have not yet been grouped into a SESSION. Alternatively, we could mix PAPERS and SESSIONS at the top-level, and add icons, colouring and/or sorting to avoid confusing the user. In both cases, the domain model would need to be reconsidered and revised, as discussed in Section 7.5.

The second direction of extending the design, is concerned with making a complete application for a certain look & feel or class of applications. First, the model of valid structures of interaction objects and windows, like the one shown in Figure 95 on page 122, may require the addition of an outer frame window containing a title and a menu bar with certain standard menus and menu items. Second, and more important, is choosing an appropriate class of application for our tasks, and conforming to its often tacit rules. In our case, we may choose to use the “Document” metaphor/application class, where the Document dialogue domain concept corresponds to the workshop task domain concept (which should be added to the domain model). Standard elements that must be included in the interface include a top-level frame containing a menu bar with the standard “File” menu. The “File” menu in turn must include the standard entries “New...”, for creating new workshops, “Open”, for managing an existing workshop, “Save” and “Save as” for saving the current state, etc.

Figure 137 shows a model that captures many of the concepts required by the DOCUMENT INTERFACE class of applications, which must be included in our design. The WORKSHOP concept has been included and related to the DOCUMENT concept, to illustrate how this model may be related to the domain model.

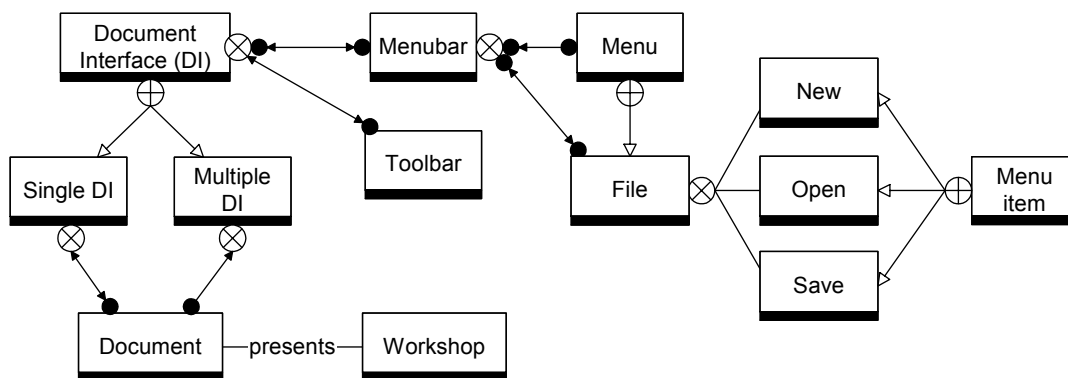


Figure 137. “Document” interface and related concepts

## 7.10 Conclusion

In this chapter we have shown how the RML, TaskMODL and DiaMODL languages may be used constructively for deriving a concrete user interface design from a task model through an abstract dialogue, with a workflow model as a starting point. By constructively we mean that all features in a model may be used to guide the subsequent design steps towards a concrete and detailed design.

With respect to the design representation classification framework presented in Chapter 3, “Design and representation”, we have illustrated several movements:

- Movement 1, from problem to solution: From each sub-task of the top-level task model we designed an abstract dialogue model. From each of these dialogue models we then considered corresponding concrete designs.
- Movement 3, from high-level to low-level: The task model was based on the processes in an existing workflow model (Figure 117), and was further decomposed into sufficient level of detail.
- Movement 4, from low-level to high-level: Several interactors in the abstract dialogue model were merged into a more complex interactor, based on the availability of a functionality rich concrete interaction objects. In addition, the dialogue models corresponding to the sub-tasks of the top-level task, were composed into a complete dialogue model.
- Movement 5, from informal to formal: Although not explicitly explained, the informal textual description was interpreted and expressed using more formal diagram-based notations.
- Movement 6, from formal to informal: To communicate our design in this document, screen shots of the concrete interaction objects were used, i.e. informal pictures representing specific classes of window toolkit elements.



# Chapter 8

## Experiences and feedback

This chapter presents the experiences we have and the feedback we have received from concrete usage of the languages, both in industry and by proof-of-concept implementation.

During our work, we have gathered experience and received feedback from practical usage of both the TaskMODL and DiaMODL modelling languages. There are three differences sources of experiences and feedback that have guided our work: external presentations and teaching, an industrial case and proof-of-concept implementations. In the following sections we will discuss these three in turn. Note that this discussion does not include the RML domain modelling language, as this language was mainly developed prior to and in parallel to the work reported in this work.

### 8.1 External presentations and teaching

Throughout the project, we have presented the representation classification framework and the task and dialogue modelling languages in two different contexts: in industrial forums and as part of (internal) teaching. In the former context, we have presented the work for professionals from software engineering and consultancy firms, at sessions within those firms, in public sessions arranged by the The Norwegian Computer Society<sup>1</sup> and as an invited presentation for a nationally funded industry-wide project on cross-platform user interface design. The presentations have ranged from shorter ones to full-day sessions.

In the teaching context, all the main parts of this work have been introduced in courses taught by the Information Systems group at our department. First, a general introduction to user interface modelling has been given in our introductory information systems course. In a second course we have focused on user interface modelling as one of a repertoire of techniques/activities that are necessary when developing interactive information systems. The design representation framework has been used to relate the models, and also to discuss the relation between user interface modelling and systems modelling, including UML, in the line suggested in Chapter 3, Section 3.4. Finally, we have given a more practical hands-on introduction to user interface design, prototyping and modelling, as part of a systems development course given to the National Office for Research Documentation, Academic and

---

1. DND (Den Norske Dataforening at [www.dnd.no](http://www.dnd.no)) is the Norwegian variant of IFIP and ACM.

Special Libraries in Norway<sup>1</sup>. This course is special since the students generally lack computer science education, although they all use computers in their daily work.

The feedback from industry has been valuable for making the modelling languages relevant and useful, and for understanding what it will take to gain acceptance for a (new) user interface modelling method, in the first place. The student's feedback has generally been useful for understanding how to communicate the approach and make the languages easier to use. The design representation classification framework presented in Chapter 3, seems to provide a useful overall vision of how different aspects of design and modelling fit together. Both industry and students stress the need for visual and flexible notations, without a formal focus. TaskMODL seems to be understandable, and both groups seem to appreciate the more expressive and graphical notation, compared to Hierarchical Task Analysis (HTA) and natural language text. The part of DiaMODL that is concerned with dataflow, i.e. the interactor abstraction, gates and connections seems to be understandable, particularly the way interactors can provide a more abstract description of concrete and atomic interaction. The way a sketch can be given an abstract interpretation in terms of interactors and be used to produce an alternative design with other concrete interactors, seems to be appreciated. When combined with Statecharts, DiaMODL is often considered too abstract, complex and "technical". The relation to and possibility of using the languages with UML has been considered important, and from this point of view our use of Statecharts has been given favourable comments.

## 8.2 The Statoil case

As a more in-depth test of the modelling languages, the author has spent between one and two days a week at Statoil's<sup>2</sup> research centre in Trondheim, during a period of three months. The context for the stay was an in-house project within the field of natural gas sales. The project participants' had a general interest in user interface modelling and some experience with modelling within the project. The project was in the process of delivering a second increment of functionality and user interface, and the focus was on how models could have been used in the analysis and design phases that already were completed.

The Statoil case was used for gaining experience with both task and dialogue modelling. The company had already used a simple workflow notation for capturing important aspects of how the gas sales department worked. When reviewing their workflow models, it became clear that the models were not complete and precise enough for outsiders to gain the necessary level of understand, without help from the developers. This problem was given two interpretations, both of which were interesting, relevant and probable, and that have been useful when working on the design of TaskMODL:

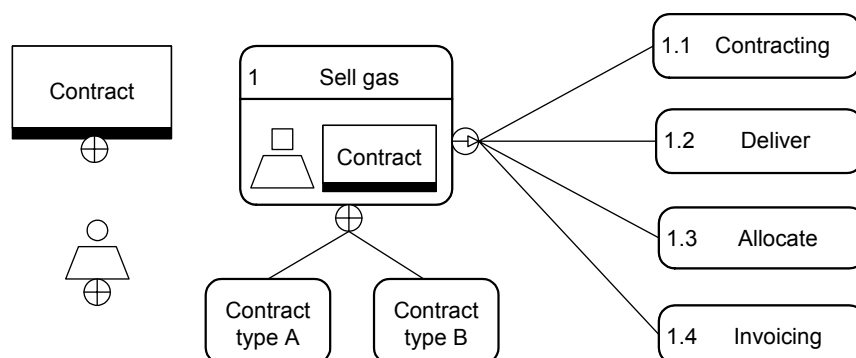
---

1. "Riksbibliotekjenesten" at [www.rbt.no](http://www.rbt.no)

2. Statoil is the national oil company in Norway

1. The models had fulfilled their needs for communication with the users and among developers, without requiring the precision and completeness that was needed for fully documenting the tasks. This supported the requirement that a task modelling language should have a flexible notation and not force the modeller to be more precise than needed.
2. The language they used was not precise enough for fully documenting the workflow and work practice. This supported the requirement that the task modelling language have a formal basis, i.e. to give the modeller the possibility of being precise and formal enough.

Another aspect of the Statoil case was important for TaskMODL's design. First, the domain model was complex and had a strong influence on how tasks were performed. Second, it was unclear which concepts were inherent in the domain and which ones were due to how tasks were performed in practice, or whether this distinction mattered at all. For instance, there were different ways of handling the various sales contract classes, and it seemed that the basis for the classification was mostly due to the difference in handling them. The model in Figure 138 is taken from the Statoil case and illustrates this situation. There are several classes of both contracts and users, and these give rise to different ways of performing the SELL GAS task. The need for handling this kind of situation prompted our work in task classification, and was part of the reason for trying to define TaskMODL in terms of RML, since RML already was strong on classification.



**Figure 138.** The task of selling gas depends on the user and on the class of contract

An early version of TaskMODL language was well received in informal presentations, and the constructs seemed to be understood by the developers<sup>1</sup> when exemplified within their project's domain of gas sales. The language was considered expressive enough for replacing the language they had used themselves, but it was difficult to judge what impact the use of TaskMODL could have on the models' precision and/or how this could have facilitated communication and documentation of the knowledge.

The Statoil case was also used for gaining experience with our dialogue modelling language. The gas sales application, was a database-intense three-tiered application, with a Visual Basic user interface. The startup screen was used for navigating among a large number of task-oriented screens and was structured as according to the workflow model.

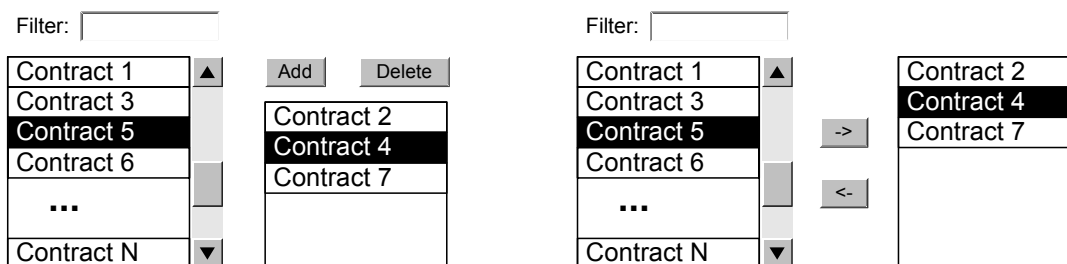
1. The language was not discussed with the domain experts or end-users (these were partly the same).

The current task, as defined by a monthly fiscal cycle, was automatically given focus, and the user could choose among the set of subtasks, or navigate to a different main task. This screen was in practice a two-level menu, although it was implemented in terms of a set of dropdown listboxes with a waterfall layout. This was an immediate example of a very explicit relation between the task and dialogue model, since the task model could be considered the domain of the startup screen. Based on our generic task model, as presented in Chapter 4, Section 4.7.1, and our models of standard interactors from Chapter 5, Section 5.2.3, we could suggest using a folder view instead. The advantage would be a more generic interface, that could more easily cope with changes in the task model, and a more compact layout, although the waterfall task structure would not be visually indicated.

The gas sales application was far too complex to be completely modelled without access to powerful tool support, so instead we selected parts that were particularly interesting, e.g. because of importance or potential for reuse. The process was similar to cycling through Nonaka's knowledge creation transfer modes. First, we externalised our understanding of the implementation in a dialogue model. Then we combined it with our knowledge of the relation between interactors and standard concrete interaction objects. The result was then presented i.e. internalised and discussed with the developers to reach a common understanding of the models. There were two noteworthy recurring observations/discoveries:

- Based on the abstract interactor model of a concrete dialogue/interaction object, we could suggest a different and often more standard/common way of implementing the same functionality.
- By comparing models of different parts of the application, we could identify inconsistencies in the implementation, in the sense that the same abstract functionality was implemented differently.

In both cases, the interactor model had made us more conscious of the abstract features of the dialogue. In both cases, it let us improve the application by selecting the most appropriate implementation in terms of concrete interaction objects. In the second case, the inconsistencies were usually due to different developers influencing the design, rather than (subtle) differences in the design problem.

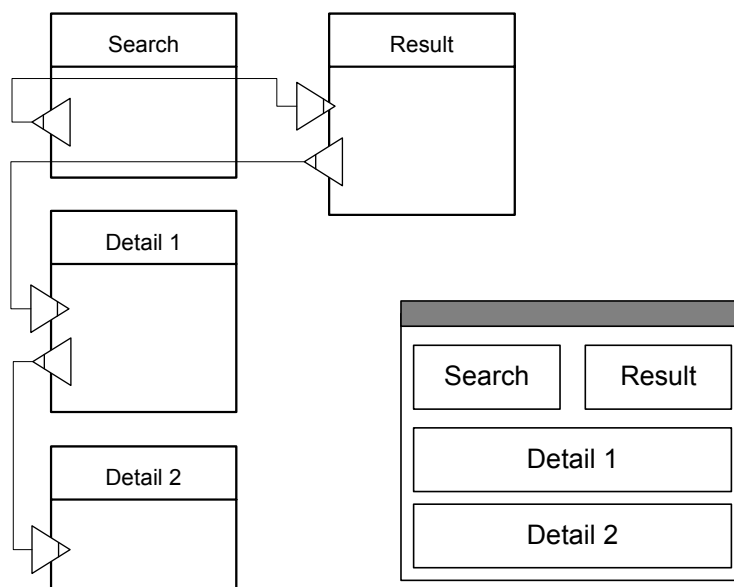


**Figure 139.** Alternative dialogues for selecting elements from a large set.  
*Left:* Adding to and deleting from selection. *Right:* Moving between lists.

In one such case there were different ways of performing multiple selection, i.e. subset of set, from structurally similar sets of elements (see model in Figure 64 on page 94). The two alternative designs are illustrated in Figure 139. Since the main set was very large, both dialogues were based on a listbox at the left containing the main set, augmented with a filter mechanism. A second listbox at the right contained the selected subset. However, one

implementation was based on *copying* elements from the left listbox to the right one, and *deleting* elements in the selection, in case of error. The other implementation was based on *moving* elements between the left and right listboxes. Being able to identify that these designs were functionally equivalent was very useful, but it did also point to the fact that the interactor model did not help us decide which variant to use. Since the decomposed interactor structure of both variants were similar, we were not able to argue one way or the other, based on model characteristics. We eventually decided that the second variant did not work well with the search/filter mechanism.<sup>1</sup> In any case it would be an improvement to settle on a single design to use in all places where this functionality was needed.

The identification of such inconsistencies in design within a single (albeit complex) application, naturally led to considering dialogue modelling as a means for design knowledge management and reuse. Statoil had already written a design guideline document for internal use, and we had already begun working on model-based user interface design patterns. Figure 140 shows an example of a highly reusable dialogue structure, derived from the Statoil case. Without the models, this kind of structure would be difficult to describe precisely. This particular model was accompanied with a Statecharts model which described the state of each part, e.g. when there was anything to show, when an element could be viewed in detail or deleted, etc. In hindsight, this pointed towards our current integrated version of interactors and Statecharts, which at that point were used separately.



**Figure 140.** Dialogue and layout pattern from Statoil case.  
Above left: Interactor structure. Below right: Layout structure.

Summarised, the Statoil case helped us gain valuable experience with using the modelling languages in general, and in particular to put focus on making the task and dialogue modelling languages and notation more flexible and easier to use. The industrial context suggested a shift of focus from formal reasoning and analysis to documentation of design and

1. It was possible, in fact easy and common, to get the dialogue into a state where an element in the right listbox could be moved to the left listbox, and at the same time disappearing because of the filter. This would happen if the filter had been changed to exclude this element, and this situation could be difficult to interpret. For instance, the filter in the right design in Figure 139, could be changed to exclude contracts with even number, and “Contract 4” could be moved from the right listbox to the left.

management of design knowledge. This is consistent with the feedback from most presentations: Most organisations are not ready to introduce formal modelling in user interface design, but nevertheless they need a notation for describing, communicating and discussing user interface design. Part of this need is experienced in the context of UML, and the lack of support for user interface design in UML-based methods, as discussed in Chapter 2, “State of the Art”. The rich graphical notation of our modelling languages seems to be appreciated, since this is in accordance with UML’s own objectives. However, the context of UML is also the main source of criticism: Since UML is becoming so popular in industry, why is our work not based on UML diagrams? We will return to this question in Chapter 9, “Conclusion and Future work”.

## 8.3 Tools and implementations

The results reported in this work are mainly theoretical. We have nevertheless built several prototype tools, which will be briefly described in this section. The implementation of these prototypes had two different goals:

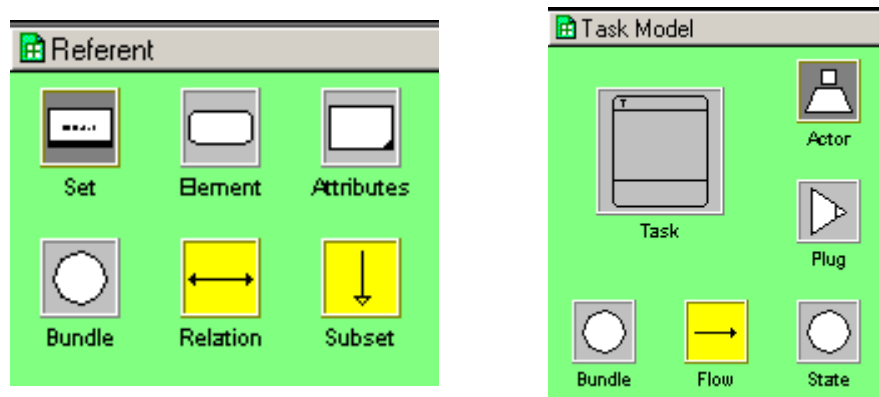
- We wanted to support drawing diagrams in our TaskMODL and DiaMODL notations, both to make them accessible to students and industry and to experiment with different notations.
- We wanted to check the feasibility and potential advantage of using the interactor language as an implementation tool, both for visual editing of user interface look & feel and behaviour, and for programming.

The former goal has been addressed with the Microsoft Visio diagramming tool. To validate the feasibility of using the DiaMODL language and notation for implementing user interfaces, we have built two kinds of proof-of-concept prototypes. First, we have implemented two prototype GUI-builders using two Common Lisp-based toolkits. Second, we have implemented a Statecharts machinery in Java, and used it to implement basic direct manipulation tools for a model editor. These efforts will be described in the following sections.

### 8.3.1 Designing the notation and drawing diagrams with Visio

The support for experimenting with the notation and for drawing diagrams has been achieved by using the Microsoft Visio diagramming tool. Visio allows the language designer to define sets of custom shapes in “stencil” files, which are made available for the modeller in palettes as dropable objects. Both 2-dimensional shapes and 1-dimensional relations can be defined, and the visual appearance can be programmed to depend on custom properties. For instance, the “bundle” circles that are used for aggregation, specialisation and element-of relations, have a type property that determines the symbol that is drawn inside it. The end-user may change properties by interacting with the shapes in three ways: through a contextual menu, by double-clicking on the shape or by invoking a popup fill-in form with an interaction objects for each property. In our RML stencil, the type of the bundle circles may be changed by using the contextual menu, while the TaskMODL stencil lets

the user change the name of a task through the properties fill-in form. More advanced functionality and behaviour can be achieved by using Visual Basic for Applications, but we have not tried this. All the models in this work have been drawn, using the stencils for RML, TaskMODL and DiaMODL shown in Figure 141.



**Figure 141.** Visio stencils for RML (left) and TaskMODL and DiaMODL (right)

Visio is an interesting alternative to both CASE tools like Rational Rose<sup>1</sup> and drawing tools like Powerpoint, because of its flexibility. For instance, Visio gives the user more freedom than CASE tools to mix formal notations with informal annotations, use text and colouring to highlight elements and include graphics. With properly programmed stencils, diagrams are quickly drawn using direct manipulation. The downside is of course the lack of support for structural constraints and semantic editing operations. Initially, we planned to modify a dedicated APM diagram editing tool, but found that the effort needed did not match the benefit. It takes a lot of resources to build and maintain user-friendly custom editors, and the Visio stencils provide a good user experience for a small cost. In addition, Visio's flexibility is particularly important in the exploratory phase, before the notation is decided upon. Based on our experience and the reasoning in Chapter 3, "Design and representation", we believe Visio is a very valuable (and undervalued) tool for both language developers and modellers.

### 8.3.2 Garnet and CLIM-based GUI-builder prototypes

To validate the feasibility of using the interactor abstraction and DiaMODL notation for implementing user interfaces, we have built two prototype GUI-builders. The prototypes have been implemented using the CLIM [Rao, 1991] and Garnet [Myers, 1990b] Common Lisp-based toolkits, respectively.

Both GUI-builders provided support for drawing hierarchical structures of interactors with connected gates. Model execution was implemented by propagating values along connections and invoking methods on the gate and interactor objects. Since CLIM is nicely integrated with the Common Lisp's type system,<sup>2</sup> the CLIM version focused on making it easier to build interactor structures by reasoning about gate types. Examples of functions include:

1. A professional version of Visio includes stencils for UML diagrams, and tools for analysing and generating code stubs.
2. We used the Common Lisp Object System (CLOS), an advanced and flexible class-based object system with multiple inheritance and runtime support for changing objects and classes.

- **Instantiation:** Many interactor structures were reoccurring, so support for templates with type parameters were added, for quickly inserting interactor structures or decomposing interactors. The type parameters could be complex, e.g. records or sequences, and corresponding interactor structures could be generated by iterating across elements of the type parameter expressions.
- **Composition:** The gates for an interactor composition could be automatically computed, based on internal interactor structure. We found it easier to modify computed structures, than building them manually from scratch.
- **Connections:** Instead of manually connecting several gates on two interactors, we supported “smart” connections, whereby the most likely set of connections between two interactors were automatically inserted. The modeller could cycle through a set of suggestions sorted by “likelihood”.
- **Functions:** When inserting connections between differently typed gates, conversion functions and filters on these connections, would be automatically inserted. I.e. instead of rejecting a seemingly contradicting (type mismatch) model, we tried to correct it.

We found that a lot could be done to simplify and reduce the effort needed for building interactor structures, based on analysis of syntax and types that would be feasible to implement in less flexible languages than Common Lisp.

The Garnet version<sup>1</sup> instead focused on linking interactors and gates to concrete interaction objects and support for mixing model editing and use. Several functions were supported:

- **Abstract to concrete mapping:** The user could select an interactor with typed gates and get a menu of the concrete interaction objects implementing the desired interface. To explore design alternatives, the user could cycle through different alternative concrete interactions objects for an interactor.
- **Model views:** The modeller could choose to see only the abstract interactor view, and edit the model. He could select a more concrete view, where the (currently chosen) concrete interaction object was shown within the interactor symbol, and hide the interactor notation altogether, and only show the concrete interaction objects.
- **Modes of operation:** Independently of whether the abstract interactor notation was visible or not, the standard behaviour of the interaction objects could be turned on or off. When on, the user interface could be tested, and if the interactor notation was visible, the propagation of values would be animated, and the values could be inspected. Note that the model could still be edited, but care had to be taken to interact with the model instead of the running interface. When interaction was turned off, it was easier to edit the model or change the layout of objects.

As a proof-of-concept, both GUI-builders showed the advantage of using an interactor model for supporting the implementation, e.g. for building the interface using templates, for

---

1. Garnet uses a tailor-made prototype/instance-based object system for its interaction objects, which are very powerful for making dynamic GUIs but more difficult to integrate with Common Lisp applications.



exploring design alternatives, and for runtime modification of the user interface. Unfortunately, the CLIM and Garnet implementation platforms are no longer useful starting points for a real implementation, hence we have turned to the Java language and platform.

### 8.3.3 Java-implementation of Statecharts for direct manipulation

The Java-based Statecharts implementation and direct manipulation tools were implemented for the purpose of validating the ideas presented in [Tr etteberg, 1998]. To support direct manipulation in a simple RML model editor, the basic Statecharts machinery was augmented by classes for geometric computation and visual objects and wrappers for linking Statecharts' events and actions to Java's event and display system. The most common editing tools, e.g. making new concepts, naming them, moving and resizing boxes, adding bundles and relations, were first modelled and then hand-translated to executable Java-classes. The tools sub-system of the editor was itself a Statecharts state, with the individual tools as substates. During runtime, the composite state was interpreted, driven by events from the mouse and generating visual feedback by means of the mouse pointer and visual objects. For debugging purposes, a hierarchical view of the state machine could be shown in a separate pane, with the active (sub)states highlighted during execution.

The advantage of the Statecharts-based implementation of direct manipulation tools, is mostly related to our own ability to analyse and gain confidence in a design, before coding it. The coding is complex, but nevertheless straight-forward in the sense that the translation from model to code is step-by-step. On a larger scale, we expect it to be easier to combine separately designed composite states machines, into more complex tools. The implementation of tooltips illustrates the flexibility of our Statecharts-based approach. Instead of separately adding tooltips handling to each tool in the palette, a tooltips (super)state was and-composed with the tools sub-system, and provided the tips by inspecting the current tool state and waiting for mouse and timeout events.

Although the Statecharts machinery is heavy-weight compared to directly implementing the same behaviour in Java component classes, it was fast enough for implementing the required feedback.<sup>1</sup> Even when animating the Statecharts machinery for debugging purposes, the response was good enough. As we consider Java a relevant platform, we have begun implementing runtime support for the hybrid interactor and Statecharts DiaMODL language presented in Chapter 5, "Dialogue modelling", on top of the Swing toolkit.

## 8.4 Concluding remarks

The contact with industry and practical experiences with proof of concept prototyping has given us confidence that our approach is worth future work. The industry has confirmed a need for:

---

1. The implementation was developed and tested on a 200Mhz laptop running Windows 95 and Sun's JDK 1.1.8.

- a flexible notation for expressing task structures found in current practices and as envisioned by designers of future interfaces.
- an abstract and graphic dialogue notation which complements the concrete (and graphic) representation that GUI-builders and development tools provide
- methods for representing, analysing and reusing design (knowledge)

However, industry remains sceptic to languages that focus more on formal characteristics than on ease of understanding and use. Hence, it seems important to pay attention to pragmatic aspects of modelling languages and the implementation of supporting tools. As for the latter, our practical experience has shown that the proposed abstract dialogue modelling language may provide a basis for practical design tools, both as the design language and as a means for implementing them.

## Chapter 9

# Conclusion and Future work

This chapter summarizes the main contribution of the work and point out directions for future research.

The motivation for this work has been our observation that different traditions or approaches to development of information systems and applications have different strengths and weaknesses. Similarly, different research fields focus on and view important aspects of information system development differently, each contributing to our understanding of the central problems, while at the same time fragmenting it. We have characterised the difference between what we have labelled *engineering* and *designer* approaches. The difference is among others concerned with the role and use of design representations, each with their strengths and weaknesses with respect to the development process.

With pros and cons for both engineering and designer approaches and for different design representations, we have focused on understanding how they (might) fit together, as formulated in our problem statement:

*How can models be used as a representation tool in the development of interactive systems and provide a means of integrating information systems development with user interface design?*

We have developed a framework for classifying design representations, where different representations populate a three-dimensional space. Movements in this space correspond to moving from one representation to another, and by performing movements according to the needs of the process, we may better utilise the strengths of each representation. The design representation classification framework and the movements within the representation space, has been used as a guidance for the second part of our work, the development of languages for modelling tasks, abstract dialogue and concrete interaction, all based on work within the fields of both model-based user interface design and information systems.

Our task modelling language was designed as a hybrid of the APM workflow language [Carlsen, 1998] and traditional task modelling languages. This will ease the movement between workflow and task models, without preventing the integration with traditional task analysis. Our dialogue modelling language took the established interactor/gate abstraction and architecture as a starting point, and merged it with the industry standard Statecharts language. In addition, it was given a flexible graphical notation, to make it more practical for expressing user interface structure and logic. We have shown how many concrete interac-

tion objects may be described in terms of the dialogue modelling language, and have used the language for modelling two common styles of interaction.

The proposed languages are mainly considered tools for expressing ideas about design, so our focus has been on simplifying the constructs and making the notation flexible. In addition, we want the models to stimulate further work on the design, so human reasoning about the models is important. Through a case study we have shown how all this fit together in a model-based approach to user interface design: How task models may guide the development of an abstract dialogue model, and how this model may further guide the design in terms of concrete interaction objects. Although we focus on humans' design and modelling activity, the languages have a formal foundation that allows for machine reasoning. Task models can (to some extent) be checked for inconsistencies, while dialogue models can be type and syntax checked and are executable. This paves the way for model-based design tools that may both be used for modelling during design and execution for testing, validation and evaluation.

## 9.1 Main contributions

The main contributions of this work are:

- We have proposed a framework for classifying design representations, and interpreted the role and use of representations during interface development in terms of this framework. The framework contributes to an enhanced understanding of when and how different design representations should be used. The framework may also be used for evaluating existing design representations and modelling languages.
- We have developed a task modelling language called TaskMODL which combines features from workflow modelling languages and traditional task modelling languages. The language provides:
  - *An advanced domain modelling language*, RML, based on simple and powerful constructs. RML is integrated with the dynamic part of TaskMODL both conceptually and visually, and is used for defining TaskMODL itself.
  - *Comprehensive support for modelling resources*, giving uniform treatment of information and resource flow and parameterisation of tasks.
  - *Support for generalisation/specialisation*, providing support for representing general and specific task structures. The support is a direct consequence of TaskMODL's definition in terms of RML.
  - *A flexible visual syntax* supporting a hybrid of hierarchical tree structure and surface containment, data and control flow and integrating the static and dynamic aspects of the domain.
- We have developed a dialogue modelling language called DiaMODL, based on a functional abstraction of how information is mediated between user and system by elements of a user interface. The language, provides:

- *Generic functional elements* based on information mediation with good support for composition.
- *Simplified activation semantics* through integration with Statecharts.
- Integration with task modelling, through the use of the same RML and dynamic constructs similar to TaskMODL.
- *Straight-forward concrete interpretation* in terms of concrete dialogue elements, supporting a smooth transition to concrete design.
- *Parameterised interactors*, supporting generic dialogue structures.
- Models of concrete interaction, based on the dialogue modelling language:
  - elements of standard *window-based* interfaces and their composition
  - the *direct manipulation* style of interaction

## 9.2 Limitations and Future work

The work presented in this work is mainly theoretical, since the focus has been on defining modelling languages and notations and not on implementing industrial strength tools and design support. There are several research and development directions worth exploring, based on the theoretical results from chapters 3, 4 and 5 and on the prototyping efforts described in Chapter 8. First, the design representation classification framework is mostly a philosophical background for our work. It should be more directly exploited for supporting and guiding the design. Second, although care has been taken to base the modelling languages on established concepts, they may still be regarded as too theoretical for practical use. Hence, the modelling languages should be made more industry-friendly by aligning and integrating them with UML. Third, in everyday work both designers and engineers are often more concerned with hands-on bottom-up design, than drawing abstract diagrams. Therefore, we should try to integrate the model-based approach with the practical methods and tools being used by designers and engineers, instead of advocating a pure top-down model-based process.

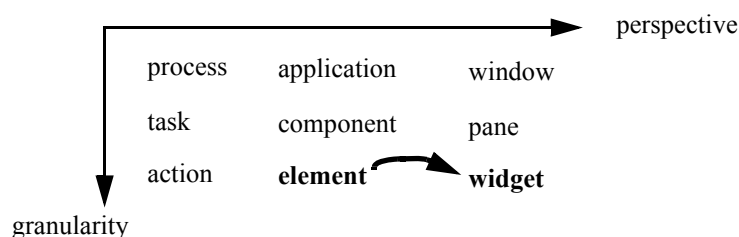
In Chapter 3, “Design and representation”, a framework for classifying design representations was presented. Based on this framework, we propose to use the TaskMODL and DiaMODL modelling languages presented in Chapter 4 and Chapter 5, respectively, to formulate design knowledge in *model-based user interface design patterns*. Our current approach will be elaborated in Section 9.2.1.

In Chapter 4, the RML and TaskMODL languages were presented. TaskMODL was integrated with and partly defined in terms of RML, and to make these more relevant for industrial usage, we propose to integrate them with UML. The advantage and potential of integrating DiaMODL with UML should be even greater, since it is more targeted at design, implementation and deployment, and is already based on Statecharts, a variant of which is part of UML. UML integration will be discussed in Section 9.2.2.

In Chapter 6, the DiaMODL language was used for describing the concrete interaction objects that user interfaces are composed from. The same interaction objects are typically provided by GUI-builders as building blocks for interface design. Based on the experiences reported in [Puerta, 1999] and [Myers, 2000], design tools based on abstract dialogue components e.g. interactors, should build on the functionality of GUI-builders. Suggestions for how abstract interactors and concrete interaction objects may be combined in a GUI-builder, will be discussed in Section 9.2.3.

## 9.2.1 Model-based design patterns

Design is about making *choices*, concerning among others *which* patterns of action the user should be able to perform and *how* the design elements are selected and composed to support this behaviour. The movements presented in Chapter 3, “Design and representation”, correspond to such design choices, e.g. how tasks are mapped to dialogue structure, how formal specifications are derived from design sketches and how dialogue structure is decomposed. Making these choices or movements requires knowledge, and accordingly, our framework can be used for classifying design knowledge. For instance, rules for mapping from abstract dialogue elements to concrete widgets correspond to the movement indicated by the arrow shown in Figure 142.



**Figure 142.** Classifying design knowledge:  
Design knowledge for mapping dialogue elements to widgets

The UID community has a long tradition of formulating design knowledge in principles, rules and guidelines, and there exists some attempt to formalize it, e.g. [Vanderdonckt, 1993]. A problem is that such knowledge is either very high-level and general or very specific [van Welie, 2000]. For capturing “middle-level” design knowledge, the use of *UID patterns* is gaining interest and momentum. The pattern concept originated in architecture ([Alexander, 1977]), and simply stated, represents a generic, proven and accepted solution to a reoccurring design problem, in a way that facilitates (re)use in a new design context. We believe patterns can become a useful design and engineering tool, if we are pragmatic about philosophical and often almost religious issues concerning pattern discovery and formulation/formats. In the context of our framework, design patterns can simply be interpreted as recipes for how to perform sound movements within our representation space [Trættemberg, 2002]. As such, they can be used bottom-up as building blocks i.e. upward movement, top-down for design refinement i.e. downward movement and to advance from problem to solution i.e. movement to the right. In a model-based approach to design, it is natural to use model fragments, and in our own experience, the abstraction and precision they provide is very helpful when formulating patterns [van Welie, 2000]. It is crucial that the modelling languages support combination of several perspectives, and this is part of our motivation for integrating them. We are currently experimenting with using model fragments using our TaskMODL and DiaMODL modelling languages in pattern formulations.

Figure 143 shows a pattern for browsing aggregation hierarchies and selecting an element. The interactor signature can be seen as a specification, and its decomposition a suggested solution to the problem of supporting this task. Two layouts are suggested for configuring them in window panes. This particular pattern concerns two movements, decomposition of dialogue and mapping from abstract to concrete design. We have formulated patterns for selecting concrete dialogue elements from abstract interactors, like “Browsing a container” shown in Figure 143, and for mapping from tasks to dialogues, like the “Managing Favourites” shown in Figure 144. The latter kind is in our experience the most difficult to discover.<sup>1</sup>

In our experience, the most immediate effect of using formal model fragments in patterns is mental, e.g. enhancing the understanding of design and increasing the consciousness of (abstract) design knowledge. However, our understanding of how design patterns should be used in the constructive design process is limited. On the other hand, model-based user interface design is an established field, and the use of model fragments in patterns may benefit both: patterns provide a new way of representing model-based design knowledge, while the use of model fragments makes it easier to use patterns in an engineering context. The use of formal models (or model fragments) in user interface design patterns is controversial, partly because of the heritage from Alexander and architecture, and partly because formal user interface models in any case are rarely used. The theory of knowledge creation presented in Section 3.6 suggests that a pattern should use both formal and informal representations, the former for precision and the latter for supporting recognition and application.

For successful use of model fragments in patterns we need gain a better understanding of:

- how model fragments may be extracted from its context, interpreted outside it and reinserted into another,
- how model fragments from different perspectives and using different languages may be related and annotated with design rationale according to the pattern idea,
- for what purpose model fragments are useful, e.g. recognition of pattern relevance vs. application of pattern,
- how the content of model fragments in terms of their classification in our representations framework may be used for structuring pattern collections and languages, and finally,
- how existing model-based tools may provide support for model-based design patterns.

### 9.2.2 UML integration

UML is a family of modelling languages that is used for both analysis and design of software systems, and has quickly become a de-facto standard within the domain of software engineering. Although it has been criticized for being system- and design-entered, rather than focused on analysis with an end-user perspective [Markopoulos, 2000a], it provides

---

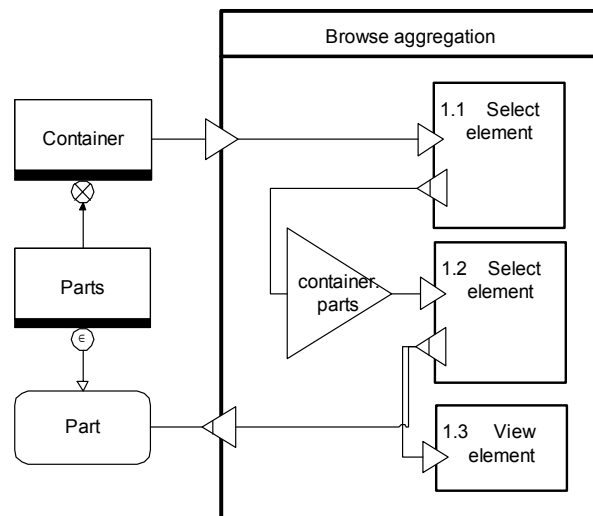
1. Our patterns employ a concise format from object-oriented design, adapted for interaction design [van Welie, 2000], while others still favor Alexander’s narrative style.

**Name:** Browsing a container

**Problem:** The user needs to browse the elements of a hierarchy and select one.

**Principle:** Provide separate connected panes for specialised viewers.

**Context:** Many application contains aggregated data, which the user must browse through. and the user often wants to invoke a function taking one of the parts as input parameter.



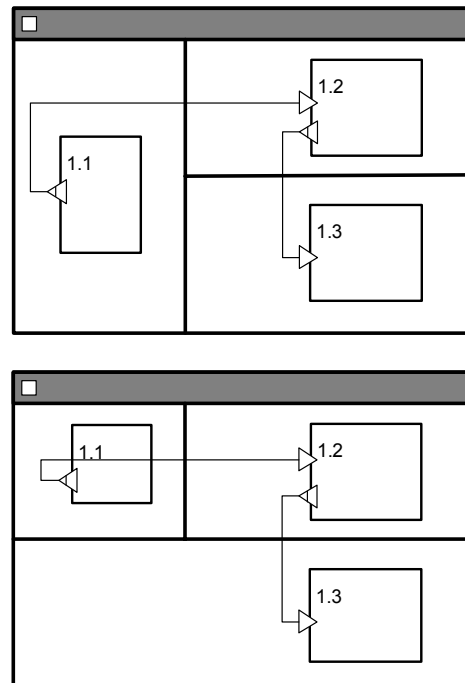
**Forces:**

- Freedom for the application to visualise the set of containers, parts and individual items in specific ways.
- Freedom for the user to resize each viewer.

**Solution:** Split a window into three panes, one for the viewing a set of containers, one for viewing a set of parts, and one for viewing individual parts. The former two must provide item selection. The selection of a container should determine the content of the parts pane, and the selected part should determine the content of the part pane.

**Examples:** Eudora email, Netscape news.

**Rationale:** The desire to act on information often comes when seeing it. Hence, it makes sense to be able to use presented information as input.



**Figure 143.** Design pattern for browsing aggregations

and opportunity for integrating software engineering and interaction design. As discussed in Section 2.4.1, UML includes sub-languages for both static and dynamic modelling that may be used within interaction design, and mechanisms for extending the existing sub-languages in case they are inadequate. Below we will outline a strategy for integrating our languages with UML.

### 9.2.2.1 RML - UML integration

The RML domain modelling language presented in Section 4.6, has been used throughout this work for two important purposes:



**Name:** Managing favourites

**Problem:** The user needs to select an item from a set. Items can be categorized as “typical” or “less used”.

**Principle:** Provide quicker interaction for the typical case.

**Context:** Many entities are so numerous that selecting from the whole set is cumbersome, while typing in is tedious. However, often some of the entities are more relevant than others.

**Forces:**

- Keeping the design simple
- Adding complexity by supporting quicker interaction

**Solution:** Let the user define which elements are more relevant or provide good rules, and provide a means for quicker selection of these. Provide a good initial set of “typicals”. Use one dialogue for the “typical” ones and one for “less used” ones, and let the user promote “less used” elements to “typical”.

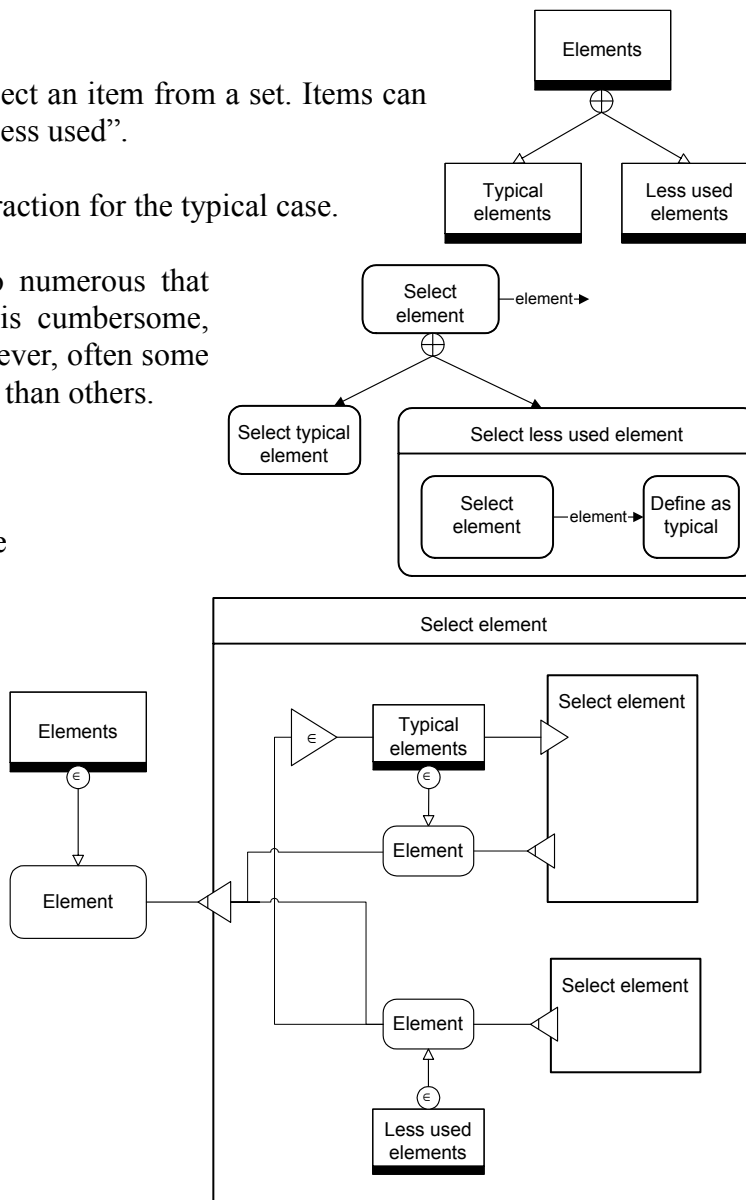
**Examples:** Colour pickers with standard colours and ability to name new ones, URL bookmarks/favourites in Netscape and Internet Explorer, Recently Used folder in Mac Finder.

**Rationale:** By streamlining the common case, overall performance is better. The user should find the simplified selection among the most used elements relieving. Remembering which elements are “typical” might be a burden.

**Figure 144.** The Managing favourites design pattern

- modelling of the concepts within the domain of task and interaction objects, i.e. domains that our languages address
- modelling of the target domain of the interface, within task and dialogue models

With such a dominant role, integrating RML and UML is the first natural step. As discussed in Section 4.6.1, RML and UML are similar in how RML elements and UML objects have



identity, and how common characteristics are described by concepts and classes, respectively. The big difference is how instances are assigned characteristics by means of classification: RML relies on set membership, while UML is based on classes as cookie-cutters for the instances' structure. It was noted that element classification has a more dynamic flavour than instantiation from a class, although this difference is not a (theoretical or practical) necessity.<sup>1</sup> If we put this philosophical difference aside and correspond UML classes and objects to RML concepts/sets and elements, most RML constructs can be defined in terms of UML's. Hence, RML can be used as a graphical notation for UML class and object diagrams, and/or UML may replace RML in TaskMODL and DiaMODL diagrams. Future research should address:

- the semantic mismatch between RML and UML and the consequences the mismatch has for different purposes, like model analysis, data management and code generation
- the pragmatic consequences of using either the RML or UML notation

### 9.2.2.2 TaskMODL - UML integration

As described in Section 4.7.1, the meaning of the TaskMODL language is partly defined in terms of a transformation to RML, and partly in terms of the life-cycle of task instances. A task definition corresponds to an RML concept, and when a task starts an instance of this concept is created and then is destroyed when the task ends. The relationship between super- and sub-tasks is defined similarly in terms of specialised part-of relations and instances: through-out the lifetime of the sub-task, a part-of relation instance between the super- and sub-task exists. The stereotype extension mechanism seems well suited for capturing these correspondences between task definition and class, and task and instance. Hence, in UML the generic task concept can be defined as a <<task>> stereotype based on the UML *classifier* meta-class, which each specific task definition will be labelled with. The aggregation relation between super- and sub-tasks may similarly be defined as a stereotype based on the special *aggregate association* meta-class. Appropriate constraints must be defined to ensure this stereotype is only used for relating <<task>> classes. A similar treatment may be given to resources, i.e. the requirement and achievement relations defined in Figure 36 on page 63.

Although this seems like a straight-forward extension of UML, several questions must be addressed in future research:

- What constraints should be put upon the various stereotypes? Should it be possible to add attributes and operation to <<task>> classes, and if so, what meaning should be given to them?

---

1. Many object-oriented *programming languages* provide means for re-classifying an instance, either by allowing direct manipulation of the is-a relation, or by providing meta-functions like CLOS' change-class method.

- Several of the generic relations in Figure 36 on page 63 are implicitly specialised when specific task hierarchies are specified. Can we handle this in UML by introducing an aggregation association class, can we introduce tags on the stereotypes to handle this, or may we just implicitly assume the specialisation without representing it in UML?
- Sequence constraints are defined in terms of the possible sequences of part-of relation instances that are created throughout a super-task's lifetime. How can these constraints be represented in OCL, UML's constraint language? Can these constraints also be useful for other kinds of relations, outside the task domain?
- A task model describes the possible sequences of tasks that may be performed, and we have defined a correspondence between task performance and the creation and destruction of (fictional) task instances. Software systems that track task performance or support task enactment, will need to explicitly represent these instances. How can basic support for this be added to the UML representation of task models. For instance, can operations be defined to check for pre-conditions, establish the requirement and achievement relations, and ensure post-conditions?

### 9.2.2.3 DiaMODL - UML integration

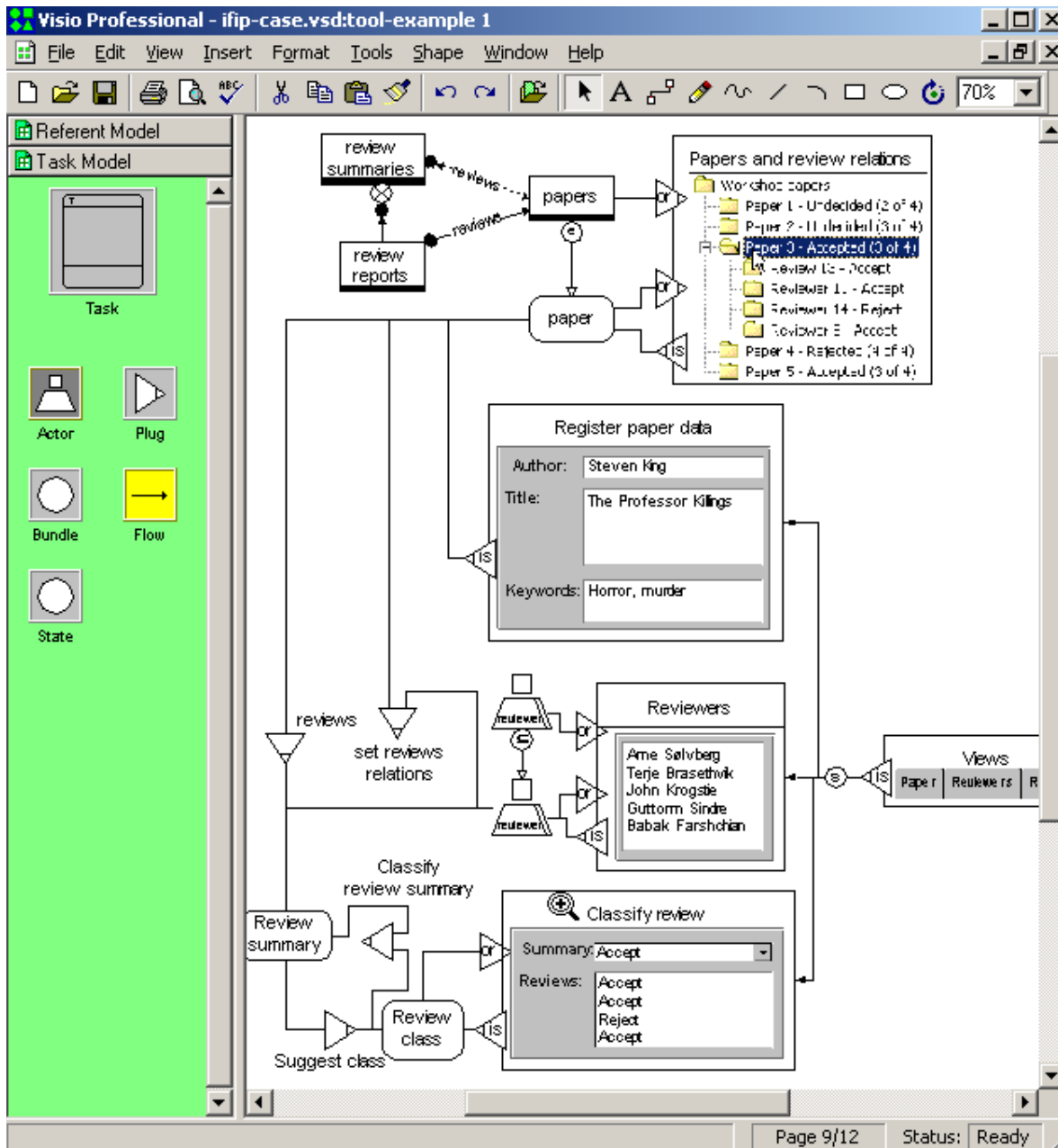
The DiaMODL dialogue modelling language has many similarities with TaskMODL: it provides a notation for defining hierarchies of process-like entities, the entities may be connected, and each entity may define a set of resources that it requires. Hence, the same strategy for UML integration as for TaskMODL may be employed: stereotypes are defined for the main concepts, and appropriate constraints are added. The <<interactor>> stereotype should be based on the *classifier* meta-class, while the <<connection>> stereotype should be based on the *association class* meta-class, to maintain the look & feel of a flow. The tip and base of gates are similar to attributes in that they have a type and can contain a value. Hence, it makes sense to define <<base>> and <<tip>> stereotypes based on the attribute meta-class, instead of a <<gate>> stereotype, and use naming to relate them to each other.<sup>1</sup>

### 9.2.3 Building User Interfaces with interactors

GUI-builders provide the user interface designer with functionality for composing a graphical user interface from concrete interaction objects, typically within a specific language and toolkit. In terms of the design representation classification framework presented in Section 3.3, this corresponds to working with design elements located far right on the solution-oriented end of the perspective axis. User interface modelling tools, on the other hand, support composing dialogues from more abstract design elements, i.e. the left part of the solution-oriented end of the perspective axis. [Myers, 2000] discuss the merits of both approaches and attribute the success of the former and lack of such of the latter to their respective low and high threshold. One approach to lowering the threshold and increasing the success of model-based tools is introducing GUI-builder functionality in them, as is done in the MOBILE system [Puerta, 1999]. It is argued that a user-centred approach

---

1. Getters and setters of properties are by convention related in a similar way: they are named by prefixing the capitalised property name with "get" and "set", respectively, e.g. "getValue" and "setValue" for the "value" property.

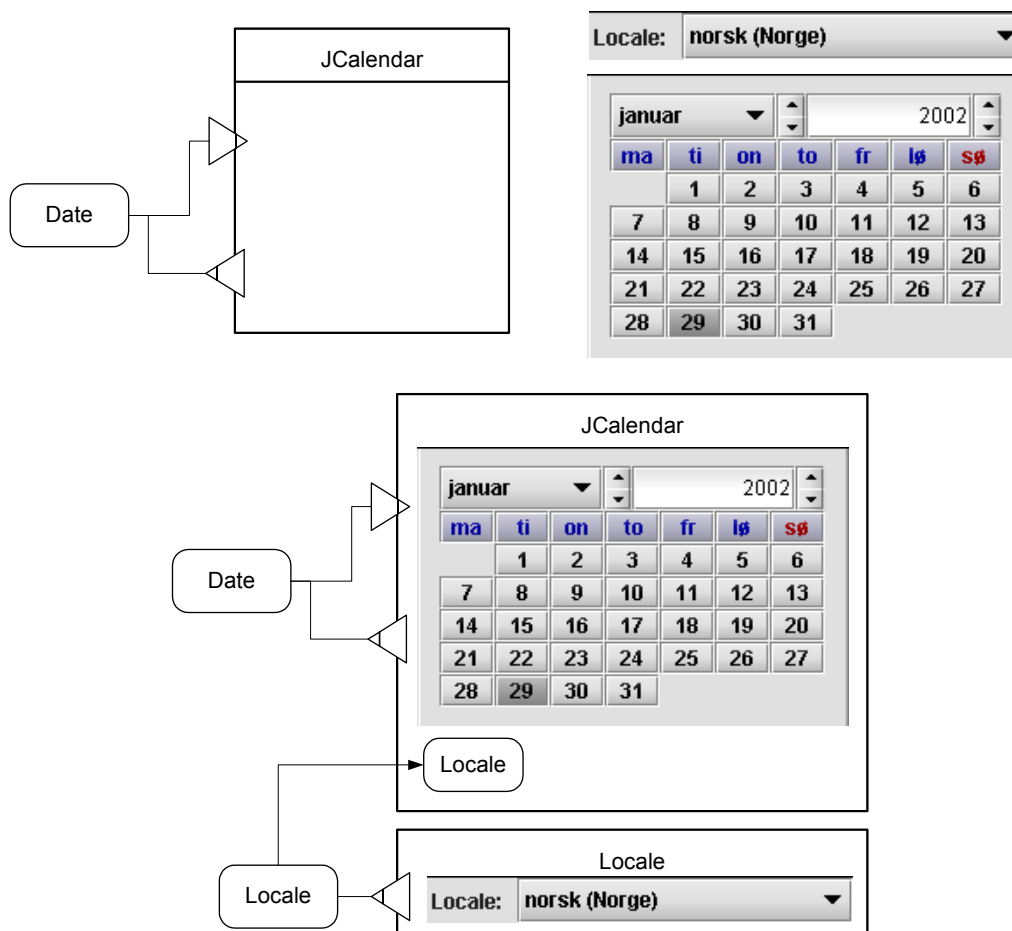


**Figure 145.** Combining abstract interactors and concrete interaction objects in one view

requires support for bottom-up design. Hence, MOBILE includes support for building the GUI from concrete elements, that later are related to abstract dialogue elements. A key point in this work is that user interface designers and developers need to move freely between different design representations, including abstract and concrete views of design elements. Therefore we propose a design tool which allows a free mix of abstract and concrete elements in a single view, a kind of integration of MOBILE's two views. Figure 145 illustrates a tool which combines interactors and their corresponding concrete interaction objects in a single view. Without the concrete interaction objects, the window could have been taken from an abstract dialogue modelling tool, while without the interactors, it could have been part of a GUI-builder. By combining these two views, we hope to gain several advantages:

- Give the designer a greater freedom in mixing abstract and concrete thinking

- Make it easier to understand the abstract notation, both for designers, developers and students
- Support both abstract evaluation and concrete testing of a design



**Figure 146.** Three different views of JCalendar interactor.

To accomplish this the tool will need to combine the functionality of both abstract dialogue modelling tools and GUI-builders, and in addition let the user manipulate the relation between abstract and concrete elements. Among others the user should be able to:

- add, edit and remove abstract and concrete elements in any order, using direct manipulation in a view similar to the one shown in Figure 145,
- establish and manipulate relations between abstract interactors and concrete interactions objects, i.e. introduce the interaction objects corresponding to an interactor as the latter's decomposition or wrap an interaction object in an interactor interface,
- for interactor devices of the kind presented in Section 5.2.2 and Section 5.2.3, choose among a set of corresponding concrete interaction objects, and for standard concrete interaction objects, introduce the corresponding interactor abstraction,
- switch among different views of (pairs of) abstract and concrete elements, as illustrated in Figure 146,

- manage libraries of (parameterised) interactors, including their decomposition
- match an interactor against a library of (possibly parameterised) interactors based on gate and resource interface, and unify them, e.g. to introduce a standard decomposition or concrete interaction object for an interactor
- define or generate prototypical data and connect it to gates of abstract interactors or set as values of concrete interaction objects
- operate the concrete interaction objects and execute the abstract interactor model, to validate/evaluate the current design

Although the tool should be usable as either an abstract modelling tool or as a standard GUI-builder, the focus should be on the integration of abstract and concrete elements, both for editing/building and executing/operating the abstract and concrete design. Figure 146 shows how a JavaBean, Java's component technology, may be modelled and viewed. In the top-left corner, an abstract view of a date device interactor is shown, perhaps as a specification of the functionality of a concrete design. Top-right a possible configuration of two JavaBeans are shown, where the bottom JCalendar bean [JCalendar@2002] shows a date in a monthly view and the top bean is used for customizing the JCalendar bean's *locale* property. The bottom view shows the interactor model of this configuration, with the actual beans included.

## 9.2.4 Summary

The three directions for future research and development outlined above, all concern how the theoretical results can be made easier to utilise and more practical and industry-friendly.

The design representation classification framework presented in Chapter 3, has in this work been used as a motivation and guideline for the development of the three modelling languages RML, TaskMODL and DiaMODL. By introducing the concept of model-based user interface design patterns, which use model fragments for capturing design knowledge, we hope to utilise the framework more directly in the design process.

During the development of the proposed modelling languages, UML has been introduced as the standard object-oriented notation and has gained widespread acceptance in the software industry. Since UML lacks support for interaction design, we have suggested a way of integrating the TaskMODL and DiaMODL languages with UML. Hopefully, this research direction may both contribute to a large community and make our own work more relevant.

A third way of utilising the theoretical results is that of building tool support for our languages. In this direction we have focused on the integration of abstract dialogue modelling with concrete design. Our proposal is a design tool that seamlessly integrates abstract interactors and concrete interaction objects in a direct manipulation interface.

# List of Figures

Figure 1. Information System (IS), User Interface (UI) and Computer System (CS) .....	1
Figure 2. People use information and tools to perform actions .....	1
Figure 4. Alternative abstract interface models, based on a task model .....	10
Figure 3. Alternative concrete interface models (CIM), based on an abstract interface model (AIM) .....	10
Figure 5. People use information and tools to perform actions .....	21
Figure 6. People, information, tools and actions are complex entities .....	22
Figure 7. Different disciplines study different parts of the picture .....	23
Figure 8. Boundaries are interesting and difficult to handle .....	23
Figure 9. The problem/solution dimension for classifying design representations .....	27
Figure 10. The level/granularity dimension for classifying design representations .....	28
Figure 11. Design representation space: perspective and level.....	29
Figure 12. The level/granularity dimension interpreted across perspectives.....	29
Figure 13. The formality dimension for classifying design representations.....	30
Figure 14. Correspondence between user and system views of the design .....	32
Figure 15. Smooth and erratic paths through the design space.....	33
Figure 16. Movements in the representation space .....	35
Figure 17. Nonaka's modes of knowledge transfer .....	37
Figure 18. The APM example model .....	44
Figure 19. The APM actor and tool symbols .....	45
Figure 20. Ontology or meta-model for task modelling languages [van Welie, 1998] .....	47
Figure 21. Task model for USER based on the workflow model example .....	50
Figure 22. Conceptual modelling constructs and notation.....	52
Figure 23. RML example model .....	53
Figure 24. Defining sex (types) as distinguishing trait .....	55
Figure 25. Named relations and membership .....	57
Figure 26. Specialisation of relations.....	57
Figure 27. Specialisation and Or-decomposition .....	59
Figure 28. Aggregation and And-decomposition.....	59
Figure 29. Combining specialisation and aggregation.....	59
Figure 30. Basic TaskMODL constructs.....	60
Figure 31. The four sequence constraints .....	60
Figure 32. Alternative task hierarchy containment style .....	60
Figure 33. The four basic task types .....	61
Figure 34. TaskMODL example model .....	61
Figure 35. TaskMODL - RML correspondence.....	63
Figure 36. Generic TaskMODL model.....	63
Figure 37. Interpreting the resource bar notation as an RML model fragment.....	64
Figure 38. Typical usage of actor modelling .....	65
Figure 39. Corresponding RML model fragment .....	65
Figure 40. Generic actor model.....	65
Figure 41. Example 1: .....	67
Figure 42. Example 2: .....	67
Figure 43. Use Phone task example .....	68
Figure 44. Decomposing set- and element-oriented tasks .....	69
Figure 45. Sequence constraints and corresponding control structure using conditions .....	70
Figure 46. Pre-condition examples .....	71
Figure 47. Task classification .....	74
Figure 48. Choice context .....	75
Figure 49. Generalising several common subtasks of using Eudora .....	76
Figure 50. Task specialisation and specialisation of resources, actors and output / post-condition .....	78
Figure 51. System-user communication.....	84

Figure 52. Mediation between system and user .....	85
Figure 53. User interface only .....	85
Figure 54. Generic interactor .....	85
Figure 55. Integer interactor composition .....	87
Figure 56. Interactor concepts .....	88
Figure 57. Summary of the notation for interactors, connections and functions .....	89
Figure 58. Functions: standalone (left), on connections (middle) and on gates (right) .....	90
Figure 59. Boolean interactor device (left) and the corresponding concrete interaction object (right) .....	91
Figure 60. Other concrete interaction objects implementing the boolean interaction device .....	92
Figure 61. Various special purpose interaction objects .....	92
Figure 62. Element selection interactor (left) and two corresponding concrete interaction objects (right) .....	93
Figure 63. Alternative selection interactor .....	93
Figure 64. Subset selection interactor .....	94
Figure 65. Listbox subset selection .....	94
Figure 66. Selecting a leaf element from a hierarchy .....	94
Figure 67. State-oriented interactor concepts .....	96
Figure 68. Main Statechart constructs .....	96
Figure 69. Selection transition .....	97
Figure 70. The checkbox behaviour .....	98
Figure 71. Selection of related element .....	99
Figure 72. And- (left) and or-composition (right) .....	99
Figure 73. Standalone functions .....	101
Figure 74. COM-like object .....	101
Figure 75. Interactor state resources .....	101
Figure 76. Mouse and pen models .....	102
Figure 77. Problem of substitution: .....	104
Figure 78. Generic string parsing and unparsing interactor .....	104
Figure 79. Interactor instantiation, through the use of resource binding .....	104
Figure 80. Interactor customisation .....	105
Figure 81. The four components of the Arch model .....	106
Figure 82. The Retrieve Messages act modelled as a torn-apart function .....	107
Figure 83. Act input device .....	107
Figure 84. Act as set and element .....	107
Figure 85. Act specialisation, based on task, argument type and result .....	108
Figure 86. The effect of different transition triggering priorities .....	110
Figure 87. Cascading acts .....	110
Figure 88. Function usage: input arguments, trigger, monitoring and result output .....	111
Figure 89. Function decomposed into three main states .....	111
Figure 90. Use (left) and decomposition (right) of Retrieve Messages function .....	112
Figure 91. Propagation of along connection .....	114
Figure 92. Combining task and interactor models .....	116
Figure 93. Using a text label for presenting numbers .....	120
Figure 94. Dialogue container and element parts .....	122
Figure 95. Frame and pane containers .....	122
Figure 96. The text label concept and interactor .....	123
Figure 97. Button structure and states .....	125
Figure 98. Boolean toggle (left) and act button (right) .....	126
Figure 99. A popup calendar .....	127
Figure 100. Popup selection and menu item interactors .....	128
Figure 101. Menu aggregation/part structure .....	129
Figure 102. Integer (within interval) device .....	130
Figure 103. Composing an input device for a set of fields .....	131
Figure 104. JTree interactor with renderer interactor parameter .....	132
Figure 105. Task and dialogue model for moving messages among mailboxes .....	133
Figure 106. The two Listbox designs: Left: dialogue box, right: wizard style .....	134
Figure 107. Aspects and elements of gesture recognisers .....	138
Figure 108. Mouse device .....	138
Figure 109. The structure of the five gesture steps .....	139
Figure 110. The behaviour of the five gesture steps .....	139
Figure 111. The Move File to Folder gesture recogniser .....	140



Figure 112. Move File to Folder interactor .....	142
Figure 113. The Visio 5 drawing tools (top) and interactor model of toolbar functionality .....	143
Figure 114. Composition of select, move and resize modes .....	144
Figure 115. Workflow process and task hierarchy alignment .....	148
Figure 116. IFIP conference arrangement problem definition .....	149
Figure 117. The process submodel from Figure 7.8, page 231 in [Carlsen, 1997] .....	150
Figure 118. Sequence of set-processing tasks .....	151
Figure 119. Aggregation of individual task sequences .....	151
Figure 120. Conceptual model of the static domain .....	153
Figure 121. Task A.2: Record response .....	154
Figure 122. Task A.3: Choosing reviewers for a paper and sending the review report .....	154
Figure 123. Refining the Review concept .....	155
Figure 124. Task A.5: Collecting the review results and making a review summary .....	156
Figure 125. Registering a paper .....	158
Figure 126. Assigning reviewers to papers .....	159
Figure 127. Alternative design for assigning reviewers to papers .....	159
Figure 128. Classifying review summaries .....	161
Figure 129. Classification as a state transition .....	161
Figure 130. And-composition of common interactor with sets of task-oriented interactors .....	163
Figure 131. Folder view of papers and related reviews .....	164
Figure 132. Tabs of tabbed pane .....	165
Figure 136. Folder view of papers and review reports provides a context for “Reviews” tab .....	165
Figure 133. The “Paper” tab: .....	166
Figure 134. The “Reviewers” tab: .....	166
Figure 135. The “Reviews” tab: .....	166
Figure 137. “Document” interface and related concepts .....	167
Figure 138. The task of selling gas depends on the user and on the class of contract .....	171
Figure 139. Alternative dialogues for selecting elements from a large set .....	172
Figure 140. Dialogue and layout pattern from Statoil case .....	173
Figure 141. Visio stencils for RML (left) and TaskMODL and DiaMODL (right) .....	175
Figure 142. Classifying design knowledge: .....	182
Figure 143. Design pattern for browsing aggregations .....	184
Figure 144. The Managing favourites design pattern .....	185
Figure 145. Combining abstract interactors and concrete interaction objects in one view .....	188
Figure 146. Three different views of JCalendar interactor .....	189

## References

- (Alexander, 1977) Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. *A Pattern Language*. Oxford University Press, 1977.
- (Alexander, 1979) Alexander, C. *The Timeless Way of Building*. Oxford University Press, 1979.
- (Arias, 1997) Arias, E., Eden, H., Fischer G. *Enhancing Communication, Facilitating Shared Understanding and Creating Better Artifacts By Integrating Physical and Computational Media for Design*. In Proceedings of the conference on Designing interactive systems: processes, practices, methods, and techniques, August 1997, Netherland.
- (Artim, 1998) Artim, J., van Harmelen, M., Butler, K., Gulliksen, J., Henderson, A., Kovacevic, S., Lu, S., Overmyer, S., Reaux, R., Roberts, D., Tarby, J-C., Vander Linden, K. *Incorporating Work, Process And Task Analysis Into Commercial And Industrial Object-Oriented Systems Development*. Workshop report from CHI'98, SIGCHI Bulletin, 4, 1998.
- (Baecker, 1995) Baecker, R.M., Grudin, J., Buxton, W.A.S., Greenberg, S. *Readings in Human-Computer Interaction: Toward the Year 2000*. Second Edition. 1995.
- (Balzert, 1995) Balzert, H., *From OOA to GUI - The JANUS System*. Nordbyn, K., Helmersen, P.H., Gilmore, D.J., Arnesen, S.A. (eds.) Proceedings of the 5th IFIP TC13 Conference on Human-Computer Interaction, pp. 319-325, 1995.
- (Balzert, 1996) Balzert, H., Hofmann, F., Kruschinski, V., Niemann, C. *The JANUS Application Development Environment - Generating More than the User Interface*. In [CADUI'02], pp. 183-206.
- (Bansler, 1989) Bansler, J. *Systems Development Research in Scandinavia: Three Theoretical Schools*. Scandinavian Journal of Information Systems, vol 1, pp. 3-22, 1989.
- (Bansler, 1993) Bansler, J.P., Bødker, K. *A reappraisal of structured analysis: design in an organizational context*. ACM Transactions on Information Systems, 11(2), 1993.

- (Bergner, 1998) Bergner, K., Rausch, A., Sihling, M. *A Critical Look upon UML 1.0*. In *The Unified Modeling Language: Technical Aspects and Applications*, p. 79-92. Martin Schader, Axel Korthaus, Physica-Verlag 1998.
- (Bier, 1995) Bier, E.A., Stone, M.C., Fishkin, K., Buxton, W., Baudel, T. *A Taxonomy of See-Through Tools*. In [Baecker, 1995].
- (Bjerknes, 1995) Bjerknes, G. Bratteteig, T. *User participation and democracy: a discussion of Scandinavian research in systems development*. Scandinavian Journal of Information Systems, 7(1):73 – 98, 1995.
- (Bodart, 1995) Bodart, F., Hennebert, A-M., Leheureux, J-M., Provot, I., Sacré, B., Vanderdonckt, J. *Towards a Systematic Building of Software Architecture: the TRIDENT Methodological Guide*. In [WfMC, compositionality], p. 262-278
- (Buxton, 1990) Buxton, W. *A Three-State Model of Graphical Input*. In [INTERACT'90], pp. 449-456.
- (Card, 1980) Card, S.K., Moran, T.P., Newell, A. *The keystroke-level model for user performance time with interactive systems*. Communications of the ACM, vol 23, p. 396-410.
- (Card, 1983) Card, S.K., Moran, T.P., Newell, A. *The psychology of human-computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates. 1983.
- (Carlsen, 1991) Carlsen, N. V. *Modelling User Interface Software*. Part I of dissertation. Technical University of Denmark. 1991
- (Carlsen, 1997) Carlsen, S., *Conceptual Modeling and Composition of Flexible Workflow Models*. Dr.Ing Thesis 1997, NTNU - Norwegian University of Science and Technology, Trondheim, Norway. Available at <http://www.informatics.sintef.no/~sca/workflow/thesis.pdf> (last visited 2002-01-17).
- (Carlsen, 1998) Carlsen, S., *Action Port Model: A Mixed Paradigm Conceptual Workflow Modeling Language*. Proceedings of CoopIS - Cooperative Information Systems '98
- (Coleman, 1992) Coleman, D. et.al. *Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design*. IEEE Transactions on Software Engineering, vol. 18, no. 1, January 1992.
- (Constantine, 2001a) Constantine, L., Lockwood, L. *Structure and Style in Use Cases for User Interface Design*. Found at [foruse.com@2001]: <http://www.foruse.com/Files/Papers/structurestyle2.pdf>

- 
- (Davis, 1995) Davis, A. M. *Object-oriented Requirements to Object-oriented Design: An East Transition?* Journal of Systems and Software. No. 30, 1995.
- (Duke, 1994) Duke, D., Faconti, G., Harrison, M., Paternó, F. *Unifying views of interactors*. In Proceedings of the workshop on Advanced visual interfaces, June 1 - 4, 1994, Bari Italy, pp. 143-152.
- (Ehn, 1993) Ehn, P. *Scandinavian design: on participation and skill*. In Schuler, D. and Namioka, A. (eds.) Participatory design : principles and practices, pages 41-78. Lawrence Erlbaum Ltd., 1993.
- (Embley, 1995) Embley, D. W., et. al. *OO Systems Analysis: Is It or Isn't It?* IEEE Software, July 1995.
- (Farshscian, 19935) Farshchian, B., Krogstie, J., Sølvsberg, A. *Integration of User Interface and Conceptual Modelling*. In Proceedings for ERCIM Workshop "Towards User Interfaces for All: Current efforts and future trends". 1995.
- (Fischer, 1993) Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., Sumner, T. *Embedding Computer-Based Critics in the Context of Design*. In [INTERCHI'93], pp. 157-164.
- (Floyd, 1987) Floyd, C. *Outline of a Paradigm Change in Software Engineering*. In Bjercknes, G., Ehn, P. and Kyng, M. (eds.), Computers and Democracy, pages 193-210. Avebury, 1987.
- (Foley, 1988) Foley, J.D., Gibbs, C., Kim, W.C. Kovacevic, S. *A Knowledge-based User Interface Management System*. In Proceedings of Human Factors in Computing Systems, p. 67-72, 1988.
- (Gross, 1996) Gross, M.D., Yi-Luen Do, E. *Ambiguous Intentions: A Paper-Like Interface for Creative Design*. In Proceedings of UIST'96, p.183-192, 1996.
- (Gaudel, 1994) Gaudel, M-C. *Formal specification techniques*. In Proceedings of the 16th international conference on Software engineering - ICSE, , p. 223-227, May 1994. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.
- (Harel, 1987) Harel, D. *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming 8, 1987.
- (Harel, 1992) Harel, D. *Biting the Silver Bullet - Towards a Brighter Future for Systems Development*. IEEE Computer, January 1992.

- 
- (Hirschheim, 1989) Hirschheim, R.A., Klein, H.K. *Four paradigms of information system development*. Communications of the ACM, 32 (10):1199-1216, 1989.
- (Horrocks, 1999) Horrocks, I. *Constructing the User Interface with Statecharts*. Addison-Wesley, 1999.
- (ISO 9241, 1997) ISO standard 9241 - *Ergonomic requirements for office work with visual display terminals (VDTs)*, 1997.
- (John, 1996) John, B.E., Kieras, D.E., *The GOMS family of user interface analysis techniques: comparison and contrast*. ACM Transactions on Computer-Human Interaction. 3, 4, p. 320-351, 1996.
- (Johnson, 1991) Johnson, H., Johnson, P., *Task knowledge structures: Psychological basis and integration into systems design*. Acta Psychologica, vol. 98, pp. 3-26.
- (Johnson, 1993) Johnson, P., Wilson, S., Markopoulos, P., Pycock, J. *ADEPT: Advanced Design Environment for Prototyping with Task Models*. In [CHI'93], p. 56, 1993.
- (Jones, 1998) Jones, S., Sapsford, J. *The role of informal representations in early design*. In [DSV-IS'98].
- (Kim, 1995) Kim, S. Interdisciplinary Cooperation. In [Baecker, 1995].
- (Krogstie, 1995) Krogstie, J., Lindland, O.L., Sindre, G. *Defining quality aspects for conceptual models*. Falkenberg, E.D., Hesse, W., Olivé, A. (eds.) Information System Concepts: Towards a consolidation of views, pp. 216-231. 1995.
- (Landay, 1995) Landay, J.A., Myers, B.A. *Interactive Sketching for the Early Stages of User Interface Design*. In Proceedings for CHI'95, v.1 p.43-50, 1995.
- (Limbourg, 2000) Limbourg, Q., Pribeanu, C. Vanderdonckt, J. *Towards Uniformed Task Models in a Model-Based Approach*. In [DSV-IS'01].
- (Lindland, 1994) Lindland, O. I., Sindre, G., Sølvsberg, A. *Understanding quality in conceptual modelling*. IEEE Software, pp. 42-49. 1994.
- (Löwgren, 1995) Löwgren, J. *Applying Design Methodology to Software Development*. In [WfMC, compositionality], pp. 87-95.
- (Malone, 1999) Malone, T.W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., Quimby, J., Osborn, C.S., Bernstein, A., Herman, G., Klein, M., O'Donnell, E. *Tools for inventing organizations: Toward*

- 
- a handbook of organizational processes*. Management Science 45(3), pp. 425-443, March 1999.
- (Markopoulos, 1994) Markopoulos, P., Wilson, S., Johnson, P. *Representation and use of task knowledge in a user interface design environment*. IEE Proceedings - Computers and Digital Techniques, vol 141, 2, p. 79-84, 1994.
- (Markopoulos, 1997) Markopoulos, P. *A compositional model for the formal specification of user interface software*. PhD thesis at Department of Computer Science, Queen Mary and Westfield College, University of London. 1997.
- (Markopoulos, 2000a) Markopoulos, P., Marijnissen, P. *UML as a representation for Interaction Design*. Presented at OZCHI 2000.
- (Markopoulos, 2000b) Markopoulos, P. *Supporting interaction design with UML, task modelling*. Position paper at [TUPIS'2000].
- (Marshak, 1997) Marshak, R.T. *Workflow: Applying Automation to Group Processes*. Coleman, D. (ed.): Groupware - Collaborative Strategies for Corporate LANs and Intranets. Prentice Hall PTR, 1997. 143-181
- (Medina-Mora, 1992) Medina-Mora, R., Winograd, T., Flores, R., Flores, F. *The Action Workflow Approach to Workflow Management Technology*. CSCW '92
- (Mukerjee, 2001) Mukerjee, A. *Interval Algebra: A review*. <http://www.cse.iitk.ac.in/~amit/interval/interval.html>, last visited 5. November 2001.
- (Myers, 1990a) Myers, B.A. *A New Model for Handling Input*. ACM Transactions on Information Systems, 8. 1990.
- (Myers, 1990b) Myers, B.A., Guise, D.A., Dannenberg, R.B., Vander Zanden, B., Kosbie, D.S., Pervin, E., Mickish, A., Marchal, P. *Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment*. IEEE Computer 23, 11. November 1990.
- (Myers, 1995) Myers, B.A., *State of the Art in User Interface Software Tools*. In [Baecker, 1995], pp 323-343.
- (Myers, 2000) Myers, B., Hudson, S.E., Pausch, R. *Past, Present and Future of User Interface Software Tools*. ACM Transactions on Computer-Human Interaction, Vol. 7, no. 1, March, 2000.
- (Nielsen, 1993) Nielsen, J. *Usability Engineering*. Academic Press, 1993.

- (Nonaka, 1998) Nonaka, I., Takeushi, H. *A theory of the firm's knowledge-creation dynamics*. In "The dynamic firm. The role of technology, strategy, organization and regions." Chandler jr, A.D, Hagström, P., Søvell, Ø. (eds). Oxford University Press, 1998.
- (Norman, 1988) Norman, D. *The Design of Everyday Things*. Basic Books, 1988.
- (Nunes, 2000a) Nunes, N.J., Cunha, J.F. *Wisdom - A UML-based architecture for interactive systems*. In Proceedings of DSV-IS - Design, Specification and Verification of Interactive Systems, 2000.
- (Nunes, 2000b) Nunes, N.J., Cunha, J.F. *Towards a UML profile for interaction design: the Wisdom approach*. Third International Conference on the Unified Modeling Language (UML'2000), York, UK, October 2000.
- (Nunes, 2000b) Nunes, N.J. *Object Modeling for User-Centered Development and User Interface Design: The Wisdom Approach*. PhD thesis from Universidade da Madeira, April 2001.
- (Olson, 1990) Olson, J.R., Olson, G.M., The Growth of Cognitive Modeling in Human-Computer Interaction Since GOMS. *Human-Computer Interaction*, 1990, vol 5, pp. 221-265.
- (Paterno, 1997) Paternò, F., Mancini, C., Meniconi, S. *ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models*. Proceedings of Interact '97, Chapman & Hall (1997) 362-369
- (Paterno, 2000b) Paternò, F. *Model-Base Design and Evaluation of Interactive Applications*. Springer-Verlag, 2000.
- (Paterno, 2000b) Paternò, F. *ConcurTaskTrees and UML: how to marry them?* Position paper at [TUPIS'2000].
- (Puerta, 1996) Puerta, A.R. *The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development*. In [CADUI'02], pp. 19-35.
- (Puerta, 1997) Puerta, A.R., Maulsby, D. *Management of interface design knowledge with MOBI-D*. Proceedings of the international conference on Intelligent user interfaces, 249-252, 1997.
- (Puerta, 1999) Puerta, A.R., Cheng, E., Ou, T., Min, J. *MOBILE: user-centered interface building*. In Proceeding of the Conference on Human factors in computing systems, p. 426-433, 1999.
- (Rao, 1991) Rao, R., York, W. M., Doughty, D. *A Guided Tour of the Common Lisp Interface Manager*. In *Lisp Pointers*, 4. 1991.

- 
- (Searle, 1985) Searle, J.R. and Vanderveken, D. *Foundations of Illocutionary Logic*. Cambridge University Press, 1985.
- (Shneiderman, 1983) Shneiderman, B. *Direct Manipulation: A Step Beyond Programming Languages*. IEEE Computer, August 1983.
- (da Silva, 2000) da Silva, P.P., Paton, N.W. *UMLi: The Unified Modeling Language for Interactive Applications*. In <<UML>>2000 - The Unified Modeling Language: Advancing the Standard. 3rd International Conference on the Unified Modeling Language, York, United Kingdom, October, 2000. A. Evans, S. Kent and B. Selic (eds.). LNCS Vol 1939, p. 117-132, Springer, 2000.
- (da Silva, 2001) da Silva, P.P. *A Proposal for a LOTOS-Based Semantics for UML*. Technical Report UMCS-01-06-1, Department of Computer Science, University of Manchester, UK, June 2001. Accessible at <http://www.cs.man.ac.uk/~pinheirp/publications.html>
- (Sukaviriya, 1993) Sukaviriya, P., Foley, J.D., Griffith, T. *A second generation user interface design environment: the model and the runtime architecture*. In [CHI'93], p. 375-382, 1993.
- (Szekely, 1990) Szekely, P. *Template-based Mapping of Application Aata to Interactive Displays*. In Proceedings of the ACM Symposium on User Interface Software and Technology, 1990, p. 1-9.
- (Szekely, 1992) Szekely, P., Luo, P., Neches, R. *Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design*. In Conference proceedings on Human factors in computing systems, p. 507-515, 1992.
- (Szekely, 1995) Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J., Salcher, E. *Declarative interface models for user interface construction tools: the MASTERMIND approach*. 1995. In Proceedings of EHCI'95.
- (Szekely, 1996) Szekely, P. *Retrospective and Challenges for Model-Based Interface Development*. In [CADUI'02].
- (Sølvberg, 1999) Sølvberg A.: *Data and what they refer to*. In Chen, P.P. et al.(eds.): *Conceptual Modeling, Lecture Notes in Computer Science*, pp. 211-226. Springer Verlag, 1999.
- (Traunmüller, 1997) Traunmüller, R., Wimmer, M., *Extending the Van der Veer Approach to Cooperative Work: A Bridge between Groupware Task Analysis and Workflow Design*. DEXA- Database and Expert Systems Applications '97.



- 
- (Trætteberg, 1998) Trætteberg, H. *Modelling Direct Manipulation with Referent and Statecharts*. In [DSV-IS'98].
- (Trætteberg, 1999) Trætteberg, H. *Modelling work: Workflow and Task Modelling*. In [CADUI'99].
- (Trætteberg, 2002) Trætteberg, H. *Using User Interface Models in Design*. In [CADUI'02].
- (UML, 1998) UML Summary, Version 1.1 <http://www.rational.com/uml/html/summary/> and Statechart notation, <http://www.rational.com/uml/html/notation/notation9a.html>
- (Vanderdonckt, 1993) Vanderdonckt, J.M., Bodart, F. *Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection*. In Proceedings of INTERCHI'93, p. 424-429, 1993.
- (Vanderdonckt, 1999a) Vanderdonckt, J.M., Puerta, A.R., Introduction to Computer-Aided Design of User Interfaces, Preface of [CADUI'99].
- (Vanderdonckt, 1999b) Vanderdonckt, J., Berquin, P. *Towards a Very Large Model-Based Approach for User Interface Development*. In Proceedings of User Interfaces for Data Intensive Systems - UIDIS'99, IEEE Computing Society Press, Los Alamitos, 1999.
- (van der Veer, 1996) van der Veer, G.C., Lenting B.F., Bergevoet, B.A.J. *GTA: Groupware Task Analysis - Modeling Complexity*. Acta Psychologica, vol. 91, pp. 297-322, 1996.
- (van der Veer, 2000) van der Veer, G.C., van Welie, M. *Task Based Groupware Design - Putting theory into practice*. In Conference proceedings on Designing Interactive Systems - DIS 2000, August 2000, New York.
- (van Welie, 1998) van Welie, M., Van der Veer, G.C., Eliëns, A. *An Ontology for Task World Models*. In [DSV-IS'98] 57-70
- (van Welie, 2000) van Welie, M., van der Veer, G.C., Eliëns, A. *Patterns as Tools for UI Design*. International Workshop on Tools for Working with Guidelines, pp. 313-324. Biarritz, France, October 2000.
- (van Welie, 2000) van Welie, M., Trætteberg, H. *Interaction patterns in user interfaces*. 7th. Pattern Languages of Programs Conference, 13-16 August 2000, Allerton Park Monticello, Illinois, USA.
- (Wellner, 1989) Wellner, P.D. *Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation*. In [CHI'89].

- (Wieringa, 1989) Wieringa, R. *Three roles of conceptual models in informatin system design and use*. Falkenberg, E., Lindgren, P. (eds.) *Information System Concepts: An In-Depth Analysis*, pp. 31-51. 1989.
- (Winograd, 1986) Winograd, T. and Flores, F. *Understanding Computers and Cognition*. Addison-Wesley Publ. Co (1986)
- (WfMC, 1994) WfMC: The Workflow Reference Model , Version 1.1, Workflow Management Coalition WfMC-TC-00-1003. 1994.

## Conferences, workshops, proceedings and sites

- (CADUI'02) Ch. Kolski & J. Vanderdonckt (eds.), *Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces CADUI'2002* (Valenciennes, 15-17 May 2002), Kluwer Academics Publisher, Dordrecht, 2002.
- (CADUI'99) Vanderdonckt, J., Puerta, A.R. (eds.), *Proceedings of the Third International Conference on Computer-Aided Design of User Interfaces*, Kluwer Academic Publishers, Dordrecht, October 1999.
- (CADUI'96) Vanderdonckt, J. (ed.), *Proceedings of the Second International Conference on Computer-Aided Design of User Interfaces*. Presses Universitaires de Namur, 1996.
- (CHI'89) Conference proceedings on Human Factors in Computing Systems. May 1989.
- (CHI'93) Conference proceedings on Human factors in computing systems. April 1993.
- (DictMind, compositionality) *Dictionary of Philosophy of Mind*.  
<http://www.artsci.wustl.edu/~philos/MindDict/compositionality.html>
- (DIS'95) Symposium on Designing Interactive Systems. Conference proceedings on Designing interactive systems: processes, practices, methods, & techniques. August 23 - 25, 1995, Ann Arbor, MI USA.
- (DSV-IS'96) Palanque, P., Bastide, R. (eds.): *Proceedings DSV-IS - Design, Specification and Verification of Interactive Systems '95*, Springer-Verlag/Wien. 1995.

- (DSV-IS'98) Markopoulos, P., Johnson, P. (eds.): Proceedings of DSV-IS - Design, Specification and Verification of Interactive Systems '98, Springer-Verlag/Wien. 1998.
- (DSV-IS'01) Johnson, C. (ed): Proceedings of DSV-IS - Design, Specification, and Verification 2001, Springer 2001.
- (foruse.com: 2001)) The site of Constantine and Lockwood's company For Use. <http://www.foruse.com/>
- (INTERACT'90) Proceedings of the Conference on Human-Computer Interaction. 1990.
- (INTERCHI'93) Proceedings of the Conference on Human Factors in Computing Systems. ACM Press. 1993.
- (JCalendar: 2002) The JCalendar home page. <http://www.toedter.com/en/jcalendar/index.html>
- (TUPIS'2000)) TUPIS - Towards a UML Profile for Interactive Systems workshop at UML'2000, available at <http://math.uma.pt/tupis00/programme.html>
- (XPath'1999) XML Path Language (XPath) Version 1.0, available at <http://www.w3.org/TR/xpath>.