From Information Management to Task Management in Electronic Mail

© Juha Takkinen 2002
Dissertation No. 732
To Kirsti and Pentti
Abstract

Electronic mail (e-mail) is an under-utilised resource of information and knowledge. It could be an important part of the larger so-called organisational memory (OM)—if it were not so disorganised and fragmented. The OM contains the knowledge of the organisation’s employees, written records, and data. This thesis is about organising and managing information in, and about, e-mail so as to make it retrievable and usable for task management purposes.

The approach is user-centred and based on a conceptual model for task management. The model is designed to handle tasks that occur in the communications in an open distributed system, such as Internet e-mail. Both structured and unstructured tasks can be supported. Furthermore, the model includes management of desktop source information, which comprises the different electronically available sources in a user’s computer environment. The information from these is used in the model to sort information and thereby handle tasks and related information. Tasks are managed as conversations, that is, exchanges of messages.

We present a language called Formal Language for Conversations (FLC), based on speech act theory, which is used to organise messages and relevant information for tasks. FLC provides the container for task-related information, as well as the context for managing tasks. The use of FLC is exemplified in two scenarios: scheduling a meeting and making conference arrangements.

We describe a prototype based on the conceptual model. The prototype explicitly refines and supports the notion of threads, which are employed so as to give tasks a context. It integrates the use of FLC into the traditional threading mechanism of e-mail, in addition to matching on text in the body. An agent architecture is also described, which is used to harmonise the information in the heterogeneous desktop sources. Finally, human-readable filtering rules created by a machine learning algorithm are employed in the prototype. The prototype is evaluated with regard to its thread-matching capability, as well as the creation of usable and readable filtering rules. Both are deemed satisfactory.
Acknowledgements

I would like to take this opportunity to thank the following people, who have contributed to this thesis either directly or indirectly.

First and foremost I owe the gratitude of all my heart to my supervisor, Professor Nahid Shahmehri, who has let me do this research. Without her knowledge, patience, and whole-hearted support this work would not have been possible. Meanwhile she has, with her good hands and mind, also made our laboratory (see below) the prospering place that it is today.

Next, a special thanks goes to Associate Professor Patrick Lambris, who joined Nahid as my supervisor right after my licentiate thesis. Patrick has a great talent for hinting and commenting on things in a good and positive way, which has had influence on my way of working.

I would also like to thank Lillemor Wallgren for taking care of both the doctoral students in general, and the related administrative issues in particular. Moreover, a special cordial thanks goes to Brittany Shahmehri for proof-reading the thesis and improving my English. I also want to give kudos to the TUS group for keeping the computers running.

This work has been done in the Laboratory for Intelligent Information Systems (IISLAB), at the Department of Computer and Information Science (IDA) at Linköping University. The research has been funded by NUTEK and SSF.

The life of a doctoral student is a period of maturing: it is learning how to do research, learning how to teach, and also learning how to learn. This life is, fortunately, not always a lonely life. I thank the enjoyable company of my former and current colleagues in, and also around, the laboratory. Thanks for making work a pleasure!

[Signature]

Julia Tabber
# Contents

## Background

1 **Introduction**
1.1 Organisational memories ................................................................. 2  
1.2 Classifying office work and tasks .................................................. 4  
1.3 E-mail and task management ......................................................... 7  
1.4 Goals and contributions of the thesis work .................................. 13  
1.5 Outline of the thesis ................................................................. 15  
1.6 Conventions ............................................................................. 16  

2 **Managing Tasks Asynchronously**
2.1 What is a task? .................................................................................. 17  
2.2 What is a task in e-mail? ................................................................. 22  
2.3 Groupware products ....................................................................... 23  
2.4 Conclusions .................................................................................. 29  

3 **Internet E-Mail**
3.1 E-mail technology .......................................................................... 31  
3.2 An exploratory case study of e-mail usage ..................................... 37  
3.3 Patterns of e-mail usage ................................................................. 43  
3.4 The thread as a task in e-mail........................................................ 46  
3.5 Summary ..................................................................................... 50  

## Theory

4 **A Conceptual Model for Task Management**
4.1 Introduction .................................................................................. 53  
4.2 Selecting a modelling approach .................................................... 54  
4.3 The scheduling-a-meeting scenario ............................................. 56
CONTENTS

10 EVALUATION
10.1 Introduction ................................................................................. 195
10.2 The set-up .................................................................................... 196
10.3 The method .................................................................................. 202
10.4 Results and discussion of the evaluation ..................................... 209
10.5 Conclusions of the evaluation ..................................................... 238

Conclusion and Future Work

11 CONCLUSION
11.1 Thesis summary .......................................................................... 245
11.2 Conclusions ................................................................................. 247

12 FUTURE WORK
12.1 The conceptual model ................................................................. 251
12.2 The language ............................................................................... 252
12.3 The prototype .............................................................................. 252
12.4 Other areas .................................................................................. 256
12.5 Epilogue ...................................................................................... 257

References

Appendices

A The Making-Conference-Arrangements Scenario .......................281
B Services and Functions in E-Mail ................................................295
Chapter 1
Introduction

E-mail has become one of the most widespread methods of communication in today’s society, and one of the most popular applications on the Internet (Lyman & Varian 2000). E-mail plays a central role in the workday—and it has found its use for sporadic (spontaneous) conversation, as well as business communication. The success of e-mail has encouraged users to employ networks for accomplishing and delegating tasks (units of work to be done) in their daily worklife. E-mail is used for the development and administration of projects and courses; information is shared, conference travels are arranged and administered, and meetings are scheduled, among other things.

What makes e-mail so special? Sproull (1991) lists six characteristics that distinguish e-mail socially from other forms of communication technology. In addition to its asynchronous nature, e-mail is also fast; it is mostly text-based; multiple receivers can be addressed simultaneously; it has a “built-in external memory” of messages that can be stored and retrieved later; and this “memory” is processable by a computer, i.e., it can be searched, edited, partitioned, and shared with others. The e-mail “memory” or archive has become of greater importance in organisations. It could be an important part of the larger so-called organisational memory, or OM for short. The OM contains the knowledge of the organisation’s employees, written records, and data. However, the e-mail memory is disorganised and fragmented, and consequently difficult for the individual user to employ in the management of tasks.
While e-mail is a useful tool, it was not originally designed to support the management of tasks, e.g., by monitoring task status and allowing adequate access to task-related information.

This thesis investigates how the disorganised and fragmented information in both old and new e-mail can be organised and compiled, and subsequently made accessible as a dynamic part of the OM.

We start by presenting the concepts of OM and define knowledge in the context of an organisation. The focus is on how information is used in an organisation. We continue with listing the types of tasks that we may find in an office. Thereafter follows a discussion of the specific problem we are investigating. We identify the goal of the thesis work. A summary of the contributions and an outline of the thesis conclude the chapter.

1.1 Organisational memories

1.1.1 Classifications of organisational memories

Organisations generally make use of two kinds of knowledge: formal and informal. According to Schmitt, Krovi and Ryker (1999) the two can be distinguished based on the extent of documentation. Formal knowledge is contained in books and manuals, while informal knowledge consists of assumptions, ideas and viewpoints. Informal knowledge also encompasses culture, shared beliefs, core values, and past experiences (or contexts) in which decisions were made (ibid.).

The notion of organisational memory (OM) is used to collectively describe both the formal knowledge and the informal knowledge described above. OM is an instance of collective memory (Durkheim, in Stein 1995), composed of “individual minds that share information through the exchange of symbols” (ibid.). Most commonly OM is defined in terms of the contents of OM and the processes associated with OM.

In order to categorise the contents of OM, several classification schemes have been proposed. For example, Churchman (1981, in Stein 1995) identifies four categories. Suggestive information is information that suggests a particular course of action, predictive information “strengthens the argument for a particular course of action,” decisive information “puts an end to controversy,” and systematic information “reminds the decision maker to consider the impact of the decision on the system.” Stein (1989, in Stein 1995) classifies OMs in terms of the level of abstraction and normative orientation of the memories—see Figure 1.1 below.

Conceptually, the OM is the means by which “knowledge from the past is brought to bear on present activities” (Stein 1995). The recurring prob-
INTRODUCTION

Figure 1.1: A classification of organisational memories (Stein 1989, in Stein 1995).

lem is then how to make the knowledge in the OM operationally available and within what type and size of context (Ackerman & Mandel 1995; Johnson et al. 1999). In our case, the typical context is the task. The OM can also be used to provide a context, e.g., by filling in detailed information about the people involved in a task.

1.1.2 INFORMATION AND THE CONTEXT OF TASKS

Regarding the context of a task, it can be said to consist of information associated with other information. The information is differentiated by form, content, and quality. According to the Oxford English Dictionary (2001), information is the “informing, telling; thing told, knowledge, items of knowledge, news.” Knowledge is by the same source defined as “familiarity gained by experience; person’s range of information; a theoretical or practical understanding of; the sum of what is known.” In other words, knowledge is information interpreted by the individual.

It is common to complain about inadequacies in the information that is received. However, a large amount of information (and messages) is not the same as good information—often the decisions made become worse (Rapp 1993) and we quickly fall victim to data glut (Roszak 1994, p. 165) or information overload. A paradoxical observation is that the computer contributes to the overflow of information, at the same time that it also is the magical lens for searching, sorting, filtering, and presenting the relevant parts of the information (Shneiderman 1998).
The information must be relevant and of the “right” amount, in order to avoid information overload. This can be achieved if the information is presented in a context relevant to the individual. The individual’s current state of mind is also important (Takkinen 1997). The context can minimise the sender–receiver distance, and also any misunderstandings in the communication. For task management purposes, the context that collects the relevant information together is the task.

For example, consider a typical task performed in a networking group of people: the scheduling of a meeting. This task involves the negotiation of a meeting time and a place, in addition to distributing results (name and number of meeting participants, agenda, documents, etc.) to the people involved. When the meeting has been successfully scheduled and held, the minutes (if any) from the meeting need to be signed, archived for later retrieval, etc. We can easily see that there is a lot of information used and produced. There are documents and other sources of information that are needed and (or) produced.

We can also see that a task consists of a number of subtasks that need to be performed before the whole task is finished. In other words, the task has different states of completion. In addition to managing information, e-mail then needs to manage tasks and task-related information in order to be an effective and usable tool. In the next section we present different approaches to classifying tasks in an office.

1.2 Classifying office work and tasks

1.2.1 Dimensions of work

According to Covey (1990), tasks of “highly effective people” can be classified in a Time Management Matrix (see Figure 1.2 on p. 5). In this matrix the tasks are organised according to their importance and their degree of urgency. Covey points out that the tasks in Quadrant II (important and not urgent) need to be handled on a continuous basis, so that the number of tasks ending up in Quadrant I (important and urgent) can be minimised. The other quadrants represent tasks that are not important and consequently need not be addressed or handled at all (if one is to be “highly effective”).

On a more formal level, office work (tasks) can be classified as being either structured or unstructured (see, e.g., Woo & Lochovsky 1992). Structured tasks are those that are predictable—known in advance—and often repetitive: the routine is the same and the steps and the selections made at each step are known. Tasks handled by administrative assistants
INTRODUCTION

Tend to be more structured than those handled by managers. Other work is unstructured and include negotiations with colleagues and customers as well as business planning. Typically, unstructured tasks require the gathering of information from several sources and getting approval from different authoritative sources. Furthermore, unstructured tasks can be unpredictable. Many office tasks fall between the extremes of the structured and the unstructured and have both types of characteristics.

Closely related to the classification scheme above is the one used by Harris and Brightman (1985)—they divide office work into maintenance and cognitive tasks. The former are routine non-cognitive tasks that do not directly produce a product, e.g., typing, filing, and performing calculations. The latter are higher level mental processes that produce an end product, e.g., a report or a form. Furthermore, the maintenance tasks support the cognitive tasks.

Not all tasks are carried out using only electronic communication, such as e-mail. Often a mixture of manual (paper-based or face-to-face-based) and electronic (e-mail-based, or a combination of several electronic communications) means are employed. An example is the sending of a manuscript via postal mail and receiving comments to it electronically. Another example is the scheduling-a-meeting scenario mentioned earlier: the meeting participants can be notified using e-mail, but the meeting room must normally be checked physically for the required audio-visual and computing equipment, and also to determine if it is at all available in the first place (e.g., via a physical calendar on the door). In other cases a task which previously
was done using, e.g., postal mail may now be accomplished purely electronically. An example of this is the submission of a paper to a conference (see, e.g., Le Geut & Dupuis 1999). Finally, computer-based communication has also introduced "new" tasks or subtasks, which previously were not appearing in a corresponding manual task. For example, electronically stored information allows for reorganisation of information on a whim and for search, retrieval, and combination of information, all in a matter of seconds.

The degree of formalisation is another dimension of the classification of tasks, i.e., how much of the task has been documented. A task is fully formalised when there exist rules and descriptions for each step or subtask contained in the task. Included in the rules are checkpoints where the state of the task is well-defined. Structured tasks (see above) can also typically be formalised. The formalisation is a way of documenting and communicating a task, both to a human user and a computer. A task description language or methodology is often used for this purpose. For example, the tasks, subtasks, and jobs (or the whole project) when managing a project are formalised using the methodology of work breakdown (Lock 1996), which means the coding and scheduling of the tasks. A critical path network or a Gantt diagram can be employed to describe the (sub)tasks and their inter-relationships (ibid.). The diagram portrays a network that displays tasks and interdependencies between them, and the estimates of time it will take to complete each task before the next task can be undertaken. Accompanying the network is a statement of work describing required tasks and deliverables.¹

Formalising unstructured tasks can be difficult since they contain an unpredictable component. Unstructured tasks are often based on cooperation and trust. However, according to our definition of formalisation above, even an unstructured task can be formalised, at least to the degree that a small set of rules for the knowledge and deliverables involved can be defined.

We present our approach to the classification of office tasks and list some examples of different types of tasks in the next section.

1.2.2 EIGHT TYPES OF OFFICE TASKS

The scheme shown in Figure 1.3 below represents our approach to the classification of office tasks. The dimensions consist of manual/electronic, structured/unstructured, and formalised/non-formalised. For example, the front-left-bottom of the cube represents an unstructured, manual and formalised task (e.g., writing a grant application).

---

¹ That is, any tangible proofs that the task has been completed successfully.
The eight categories represented by the eight compartments in the figure above—and corresponding task examples—are described in Table 1.1 on p. 8. We are especially interested in the tasks that are less structured and less formalised, and which are either electronic or partly electronic and partly manual (e.g., partly paper-based).

### 1.3 E-mail and task management

#### 1.3.1 E-mail and the organisation

Consider a geographically distributed workgroup of people working together to achieve a common goal within an organisation. The people may also be distributed in time, either having working hours that overlap only partly with each other—or not at all—or they may even be living in different time zones. This is a typical scenario where an asynchronous communication tool such as e-mail is popular and, above all, usable. The people in the group need not be synchronised in time to communicate with each other and exchange messages. In addition to being usable because of its asynchronous nature of functionality, e-mail is easy to use. It is also ubiquitous, i.e., available to everyone using a computer.

As hinted at in the introduction to this chapter, there are problems with e-mail use. E-mail started out as an application for communication but is today used for several other purposes. It is used as a general tool for managing time, information, and tasks (Mackay 1988). More specifically,
**Table 1.1: Eight task types and corresponding examples**

<table>
<thead>
<tr>
<th>Type of task</th>
<th>Example</th>
<th>Examples of support and sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured, manual, formalised</td>
<td>Completing an income-tax return form</td>
<td>Preprinted form, written instructions, receipts</td>
</tr>
<tr>
<td>Structured, manual, non-formalised</td>
<td>Writing comments on an article distributed at a weekly held study circle</td>
<td>The article, pencil, paper</td>
</tr>
<tr>
<td>Structured, electronic, formalised</td>
<td>Sending an EDI (Electronic Data Interchange) message in electronic commerce</td>
<td>EDI protocol and software</td>
</tr>
<tr>
<td>Structured, electronic, non-formalised</td>
<td>Sending a call for a weekly meeting using e-mail</td>
<td>E-mail program, agenda, list of participants and e-mail addresses</td>
</tr>
<tr>
<td>Unstructured, manual, formalised</td>
<td>Writing a grant application</td>
<td>Application instructions, colleagues, previous applications</td>
</tr>
<tr>
<td>Unstructured, manual, non-formalised</td>
<td>Developing a new course which uses a new examination form</td>
<td>Books and workshops about examination forms, departmental course development procedure, colleagues</td>
</tr>
<tr>
<td>Unstructured, electronic, formalised</td>
<td>Compiling and posting an FAQ (Frequently Asked Questions) in net news</td>
<td>Previous FAQs, netiquette (Internet etiquette), net news colleagues</td>
</tr>
<tr>
<td>Unstructured, electronic, non-formalised</td>
<td>Planning a night out with friends using Internet chat</td>
<td>Chat program, friends</td>
</tr>
</tbody>
</table>

e-mail is employed for delivering documents, storing names and addresses of people, scheduling meetings, asking for assistance, and handling technical support queries (Whittaker & Sidner 1996). E-mail is indeed a flexible tool, but the adaption of e-mail for an organisation in different ways has led to problems. Whittaker and Sidner (1996) have coined the term *e-mail overload* to denote the use of e-mail for other purposes than basic communication.
1.3.2 **Problems with e-mail and tasks**

E-mail will continue to be very popular—ubiquitous—to a large degree because of its ease of use. E-mail is now the source of many different work tasks, serving as the place in which work is received and delegated (Whittaker & Sidner 1996).

The increasing employment of e-mail to accomplish tasks implies more and more problems. Users devise and employ *ad hoc* solutions for tracking and delegating tasks. For example, in order to manage tasks successfully, users are required to ensure that information relating to current tasks is readily available (Whittaker & Sidner 1996). To accomplish this, a user often may employ the ad hoc solution of mailing information not originally contained in an e-mail message to herself in order just to bring the information into the e-mail system (Fleming & Kilgour 1994).

The inbox, the special folder in most e-mail systems where incoming messages are put by default, has a central role in the ad hoc use of e-mail. For many users the inbox determines the contents of the workday. In many cases the inbox tends to become cluttered with messages.

The ad hoc solutions discussed above are a consequence of the flexibility of e-mail—e-mail can be used according to circumstances. Ad hoc solutions are rather natural, generally speaking. Depending on material and social circumstances, *every* course of action is essentially ad hoc (Suchman 1987).

Moreover, whenever computers are involved in a task there is no “natural” documentation being done, such as the storage of papers in a binder on an office shelf. An e-mail-based task and the task-related documents are both more or less “invisible” and require some effort to retrieve and examine. Only the messages belonging to a task, hopefully stored in an appropriate folder, remain.

1.3.3 **Current solutions**

*Workflow systems* (discussed more extensively in Chapter 2) address the general problem of how to support the process of organisational (and asynchronous) work (Hazemi *et al.* 1996b). Workflow works best with well-established, highly-structured tasks. *Groupware* (discussed more thoroughly in Chapter 2) on the other hand is specialised software designed for the use of collaborative workgroups (Johansen 1988), often with both synchronous and asynchronous capabilities. One popular example of a successful groupware product is *LOTUS NOTES* ("IBM/Lotus" 2001), which rather

---

2. For now, we exclude e-mail from the notion of groupware.
fully supports **collaborative work**. This is accomplished mainly through the availability of **synchronous** communication means, such as chat-like workrooms integrated with telephony based on the Internet Protocol (IP).

However, groupware systems have two major problems, as compared to solutions that build on already existing, ubiquitous e-mail-based communication in an organisation. Firstly, the systems are economically too expensive to introduce into an organisation—expensive software licenses and the required effort to adjust and develop applications to meet specific organisational needs (Hazemi *et al.* 1996b) are two examples. Secondly, to learn how to use groupware products requires extensive effort, often not on a par with the final benefits of the use of the systems.

We saw in the previous section that there is a strong need for support of task management in e-mail. What are the current solutions? Different techniques have emerged to augment the functionality of e-mail systems for task management purposes:

- **rules**, which are used to act on messages or sort messages into folders
- **control elements**, which describe how messages are processed or transferred from one user to another or within a group—the sequence of recipients in the context of some business procedure, *e.g.*, can be defined using special routing commands or forms (*cf.* Hollingsworth 1995)
- **message structures**, which organise the information within the body of the messages, for example using MIME (Freed & Borenstein 1996) or forms and templates (Malone *et al.* 1987a).

E-mail systems have been progressing towards workflow functionality through the addition of the techniques above, especially in the form of control elements. For example, Goldberg, Safran and Shapiro (1992) describe how to make messages more active or “intelligent” in a framework called **ACTIVE MAIL** (discussed in Chapter 6).

### 1.3.4 Restructuring e-mail to accommodate tasks

Organisations were turned into document management organisations in the 1980s, with an over-confidence in rules, regulation, and bureaucracy (Dahlbom 2000). Office work was to be automated and designed as routine work, along the lines of factory production. Some of the office work can still be automated. However, we now have an increased confidence and interest in the responsibility, knowledge, and initiative of individuals (*ibid.*). The view on information, knowledge, and tasks has become human-centred.

The automation of a process—and especially the **whole** process—in e-mail according to a workflow model is too rigid. Tasks accomplished
through the use of e-mail-based systems are mostly loosely structured and
of an ad hoc nature, which require a different approach with regard to sup-
port. As noted earlier, the ad hoc nature is a consequence of the flexibility
of e-mail.

An examination of task management models for e-mail reveals a dis-
crepancy between the current ad hoc use of e-mail and the “ideal” use of e-
mail for task management purposes. There is a need for the e-mail-based
communication to be coordinated, in order for tasks to be managed.

Ideally, the task context is always preserved; a user can switch between
different task contexts as well as remind herself about a task, among other
things (cf. Whittaker & Sidner 1996). Croft and Lefkowitz (1984) list the fol-
lowing eight major user requirements for services in a task support system:

1. **Automation of routine tasks**: structured tasks can be automated.
2. **Assistance with complex tasks**: actions that are needed to be per-
formed as part of a task need to be supported, in the form of explana-
tion, error detection, and correction.
3. **Context switching**: the system should facilitate switching between tasks.
4. **Handling unusual actions**: there are tasks that need to be carried out
in non-standard ways, which the system should support, e.g., either by
preventing the unusual action or directly invoking a suitable tool di-
rectly. Better still is if the system can recognise the non-standard task
and directly support that task.
5. **Adaptability**: procedures are constantly being added and modified in
an office. User-defined procedures are an important part and should be
supported, as well as means of maintaining the consistency of related
procedures.
6. **Task invocation**: users should be able to invoke tasks in a fashion
similar to that for invoking generic tools. Tasks should be at different
levels of abstraction, e.g., filling in a form vs. processing an order.
7. **Status inquiry and explanation**: a method for describing the status
of current tasks and explaining possible next steps should be provided.
8. **Handling multi-user tasks**: the system should support and coordinate
tasks distributed among a number of people. That is, it should be pos-
sible to attach people (or a role) to parts of a task.

We note that by using e-mail as the basis for task management not only
does e-mail become our **platform** but also the “transport engine” of the
services for task management.
With regard to the development of a collective knowledge in the form of an OM (see Section 1.1 on p. 2), e-mail is a rich medium, but only if provided with a context (Robertson, Sørensen & Swan 2000). This is true even when compared with a product such as LOTUS NOTES (“IBM/Lotus” 2001). The context for e-mail is, for example, the grouping of the received messages in a meaningful way. This grouping of messages also reduces the cognitive load on the users that tend to clutter their inbox—Whittaker and Sidner (1996) propose the use of conversational threading and semantic clustering to reduce this clutter. These proposed functions also support context switching (the third point in the list above) since a message is always linked to another message, which reduces misunderstandings. Whittaker and Sidner (1996) also suggest a function to mark particular inbox items as requiring action. This function is part of the task status inquiry and explanation (the second-to-last point in the list above).

Related to the marking function above is the ability to program reminders. Short tasks are easy to handle via e-mail, according to a study by Bälter (1998, p. 132). Longer tasks, on the other hand—which take more time to complete—need a reminder function. For example, consider the task of making conference arrangements with the aid of an administrator. The task consists of a number of subtasks: registering for the conference, reserving a hotel room, etc.—all of them are delegated to the administrator. There is also a deadline involved for each of the different subtasks, so a reminder function is needed to check the status of the subtasks.

Longer tasks also involve reconceptualisation, which is “the gradual evolution of human understanding during task performance” (Shipman & Marshall 1999, p. 13). Shipman and McCall (1994) advocate an approach for handling longer tasks where certain mechanisms are provided for information to be first entered in a less formal representation. These mechanisms should then be used to incrementally formalise and structure the information as time progresses. In other words, this incremental formalisation strategy seeks to reduce the overhead of entering information and defer formalisation of the information until later (Shipman & Marshall 1999). An example of a support for incremental formalisation is a system that suggests a filtering rule to be added to the system, based on a user’s reading patterns. In this example, simply by presenting the rule to the user makes her understand her own goals better and thereby overcome a potential barrier to formalisation (ibid.).
INTRODUCTION

1.3.5 TOWARDS A TASK MANAGEMENT MODEL FOR E-MAIL
In order to include support for task management in e-mail and avoid e-mail overload, a restructuring of the application domain of e-mail is needed. We need a conceptual model to display—and reason about—the required support for task management. In the model we need to incorporate both the formal and informal types of tasks, as well as the electronic and manual tasks (cf. Table 1.1 on p. 10). Furthermore, having a conceptual model, we can compare e-mail systems to each other with regard to task management capability.

The conceptual model for task management support in e-mail should be general enough to make it possible to incorporate task management into any existing e-mail system. The model should take into account the manual tasks that intersect with—and influence—the electronic environment of the e-mail user. The electronic environment—that is, sources such as old e-mail messages, calendar software, and web bookmarks—is part of the organisational memory (OM) (see Section 1.1 on p. 2). (We define the manual environment as consisting of paper-based sources such as manuals and books, as well as face-to-face communication with other people.) The model should also preserve the social aspects of e-mail usage: there is also a need to be able to communicate informally, but still within a task context. Therefore, the model should include non-formal communication (informal conversations) even if there is a structured context (formal conversations or tasks) present.

The success of Internet e-mail shows us that providing openness, that is, the ability to integrate or use other applications, is essential when modelling task management. A task management model for e-mail based on openness will provide flexibility to the user. She will have the possibility to gradually introduce—and incrementally add to—the task management functionality in her e-mail, at the same time having control over the ease of using it. Finally, since asynchronicity is one of the main characteristics of e-mail, we want to keep that intact when restructuring the application domain.

1.4 Goals and contributions of the thesis work
We have the goal of restructuring the application domain of e-mail in order to accommodate task management. We wish to give an integrated view of task management in e-mail, starting from a conceptual model. In this model we aim to handle all the types of tasks presented above, but with a focus on the ones that are less structured and less formalised. These
are the types of tasks that are elusive, and difficult to model, especially in an asynchronous system such as e-mail. We do the restructuring of e-mail by first designing the conceptual model for task management in e-mail and then developing a prototype based on the model.

The conceptual model provides us with a task-oriented and personalised (user-centered and individualised) management and view of information and knowledge. The information and knowledge is mainly available from the e-mail system itself, but is also part of and related to the organisational memory.

A Formal Language for Conversations (FLC) is defined in the thesis. The language is based on ideas from speech act theory. FLC is used as a container for task-related information in a message, and also for defining the context of a task. The use of FLC is exemplified in two scenarios: scheduling a meeting and making conference arrangements.

A prototype is implemented in order to test and evaluate central concepts of the conceptual model. The prototype is based on an existing e-mail system. The reason for this is to show that an e-mail system can be extended with new functionality according to the model. An important issue here is that we also want to keep the ease-of-use of e-mail, in addition to keeping the asynchronous nature. The prototype is evaluated from a user’s viewpoint.

The conceptual model and the prototype are open and general-purpose. That is, the model itself is possible to extend easily and it is not restricted to a specific type of task.

The work in this thesis follows the research tradition represented by mainly three related works: THE COORDINATOR, INFORMATION LENS and MMS/FLBC. THE COORDINATOR (Winograd & Flores 1986) is recognised as one of the first groupware systems. It uses the concept of conversations based on the idea of speech acts (Austin 1962; Searle 1969) to model office work and coordinate communication. INFORMATION LENS (Malone et al. 1987a) is an “intelligent system for information sharing and coordination,” which introduced the concepts of templates to achieve semi-structured e-mail messages and also user-controlled filtering. MMS/FLBC (Kimbrough & Moore 1997) is an application developed to replace communication that uses EDI (Electronic Data Interchange—a standard for messaging in electronic commerce). MMS/FLBC is—like THE COORDINATOR—also based on speech act theory, but decouples (unbundles) the conversation handling mechanism (see THE COORDINATOR above) from the application. Instead it implements this mechanism as a separate communication protocol. We
INTRODUCTION

discuss all three of these interesting systems, as well as other relevant systems, when we compare our approach with related work in Chapter 6.

1.5 Outline of the thesis

The outline of the thesis is shown graphically in Figure 1.4, with the relations between the different chapters explicitly laid out.

Figure 1.4: The relations between the chapters in the thesis.
In the following two chapters we present, step by step, the background for our work. We discuss the principle of asynchronous management of tasks in Chapter 2 and put it in the context of groupware and workflow. The characteristics of Internet e-mail are investigated in Chapter 3, where we also summarise studies of how e-mail is used today for task management purposes. Furthermore, the notion of task is defined in the e-mail context, as well as what we mean by the management of tasks. Chapter 3 concludes “Part I: Background” of the thesis.

In “Part II: Theory” we first present our conceptual model which includes task management support in e-mail (Chapter 4). Thereafter, in Chapter 5, we describe a formal language for conversations. The language is used to structure messages and also link them together, in order to provide a context for tasks. In Chapter 6, which concludes the second part of the thesis, we compare our approach with the work of others.

The third part of the thesis, “Part III: Implementation,” begins with Chapter 7, where we discuss the design considerations and the requirements for an implementation of the conceptual model presented earlier. A prototype is thereafter described from two viewpoints. The first viewpoint is the developer’s (Chapter 8) and the second viewpoint is the user’s (Chapter 9). The third part of the thesis is concluded with an evaluation of the prototype (Chapter 10).

In “Part IV: Conclusion and Future Work” we summarise our work, list the contributions, and draw conclusions from it. This is done in Chapter 11. We conclude the thesis with Chapter 12, which is about future work.

1.6 Conventions

The following conventions are used in the thesis:

- *italics* mark foreign words, phrases, new concepts introduced, abbreviations, mathematical unknowns, constants, and expressions
- SMALL CAPS mainly denote product names, including acronyms and abbreviations representing systems or products
- **bold** face is used to emphasise a word or highlight a concept
- **courier** is used for code, e-mail messages and command line examples.
Chapter 2
Managing Tasks Asynchronously

In this chapter we examine the notion of a task and its relation to workflow, computer-supported cooperative work and groupware. We examine how a task can be represented in e-mail, our example of an asynchronous communication method. We then examine the state-of-the-art of a number of popular groupware products with regard to asynchronous task management. The chapter is concluded with a discussion and a summary.

2.1 What is a task?
Generally speaking, a task is a unit of work to be done (Rusinkiewicz & Sheth 1995). A task can consist of one or more subtasks (see Figure 2.1). For example, the task of scheduling a meeting involves the subtasks of checking available times among the participants, selecting a suitable meeting time, and booking a meeting room. A task is performed by a so-called processing entity, which can be a human, a software system such as an e-mail client, a database management system (DBMS), etc. (ibid.).

A task can be specified in a number of ways, including a textual description in an e-mail message, a form or a computer program. A task can be seen as having different states. Typically a state transition diagram is used to describe a task on an abstract level (see Figure 2.2). The figure shows an information management system (IMS) transaction. In the state transition diagram different states of the task are explicitly shown. The arrows in the figure denote transitions between the different states: initial,
submitted, executing, committed, and aborted. Typically, for each transition the enabling conditions are also noted (not shown in the figure).

With regard to multiple tasks, they can have the properties of being either synchronous or asynchronous, specialised in either information exchange or information-sharing, and suited for sequential or concurrent processing, among other things. The collection of activities that involve the coordinated execution of multiple tasks by different processing entities we call a workflow (Rusinkiewicz & Sheth 1995) (see Figure 2.3). A station or a processing entity in the figure is, e.g., a workplace for authorising invoices for payment. As also shown in the figure, there can be feedback connections in

---

**Figure 2.1:** The relation between work, tasks and subtasks.

**Figure 2.2:** A state transition diagram for an IMS transaction.
Managing Tasks Asynchronously

Figure 2.3: An example workflow.

A workflow, so that a task can be sent back to a station where it already has been processed. These connections are usable in cases when there have been some uncertainties in a station. A workflow is a number of tasks performed either in series or in parallel by two or more members of a workgroup, who have a common goal. An example of a common goal could be to produce a report or to determine a meeting place and time. In short, adding time (when) and place (where) as variables for group work gives us the notion of workflow. According to the Workflow Management Coalition (“Workflow” 2001; Hollingsworth 1995), a workflow is “the computerised facilitation or automation of a business process, in whole or part.” A “business process” is then “a computerised representation or model of a process that defines both the manual process and the automatable workflow process” (“Workflow” 2001).

Workflow systems are mainly constructed for large workgroups. There exists a variety of products that have been developed to support the interaction, collaboration and messaging between group members. A common term used for these products is groupware. Other terms often heard are workflow management, workgroup computing, collaboration software, and computer-supported cooperative work (CSCW). In general, groupware is a term used by the business sector to refer to the numerous products available for workgroup support, while CSCW is a term mainly used by the research community (Hazemi et al., 1996b).

Based on whether users are working asynchronously or synchronously, at the same place, or different places, the domain of groupware or CSCW can be separated into four quadrants (see Figure 2.4) (Ellis, Gibbs, &
Relevant to this thesis is the lower right quadrant in the figure: the asynchronous distributed interaction part of groupware. This quadrant represents a special type of workflow tasks, namely those that do not require members to work together at the same time and that can take anywhere from minutes to, say, months to complete.

Compared to workflow systems, groupware is generally developed for smaller groups. There are other possible classifications (Prinz 1989). For the purpose of this thesis, we use workflow in the most general sense to denote systems that mainly model structured tasks.

A typical example of a workflow is the purchase order processing in office computing. This process is structured and can be defined beforehand, e.g., using a procedural language. A typical task in this context is the processing of forms, which can be done by humans, applications and (or) DBMSs. Forms (Gehani 1982) are among the most often used design solutions for carrying out tasks electronically. A person sends a form—a message with formatted fields that specify what is to be done, what has been agreed upon, when tasks are to be completed, etc.—to another person, asking her to do some work. The person asked replies either with acceptance, rejection, or with a modified proposal (cf. Palme 1995, p. 40). There are many examples of user interfaces that use forms as a metaphor for handling tasks—see for example Kreifelts and colleagues (1991). Electronic forms retain many of the properties of paper forms and are therefore natural to use (Gehani 1982). They are also easy to couple with a database—a form simply represents a view of the database. However, using forms for

---

**Figure 2.4: The groupware place–time relations (Johansen 1988).**

<table>
<thead>
<tr>
<th></th>
<th>Same Time</th>
<th>Different Times</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Same Place</strong></td>
<td>face-to-face interaction (meeting room systems)</td>
<td>asynchronous interaction (shared workspace, desktop video conferencing)</td>
</tr>
<tr>
<td><strong>Different Places</strong></td>
<td>synchronous distributed interaction (shared database, co-authoring systems)</td>
<td>asynchronous distributed interaction (message-based collaborative systems)</td>
</tr>
</tbody>
</table>

Managing Tasks Asynchronously

Handling unstructured tasks tends to complicate things for the user (see Chapter 1 for examples of classifying tasks).

A workflow can be specified by describing its constituent tasks. Furthermore, rules, constraints or programs can be used to describe inter-task dependencies, i.e., how the tasks should be coordinated in the workflow. Also, the requirements for executing a workflow correctly have to be defined. The purpose of this is to restrict the execution of the workflow to meet application-specific correctness criteria (Rusinkiewicz & Sheth 1995).

An example of a workflow containing both electronic and manual tasks (cf. Section 1.2 on p. 4) is shown in Figure 2.5. It shows the workflow for examining students and registering the results. Each box in the figure repre-

Figure 2.5: A workflow with both manual and electronic tasks.
sents a task, with the processing entity as the header (“Student”) and the task’s name in the body (“Do lab assignment”). The text by the arrows denotes deliverables. The dotted lines denote electronically managed communication.

A workflow management system (WfMS) automates the execution of business processes by passing documents, information or tasks from one person to another according to a process definition. The WfMS typically consists of a scheduler and a number of task agents, with one agent per task. The task agent controls the execution of a task by a processing entity (Rusinkiewicz & Sheth 1995). A WfMS is “a system that completely defines, manages and executes ‘workflows’ through the execution of software whose order of execution is driven by a computer representation of the workflow logic” (Hollingsworth 1995). Typically, a process modelling language is used to describe the flow of work from one person or IT resource to another. To execute a workflow correctly, all inter-task dependencies must be enforced. The scheduler may submit tasks for execution by a task agent or request that a previously submitted task be aborted. It also determines allowable transitions of each task based on different system and user events. In the domain of e-mail, an example of a simple use of a scheduler is a message that is automatically resent to a recipient after a predetermined amount of time, as a reminder.

The task agents may also coordinate their execution by communicating with each other to satisfy task dependencies and other workflow requirements. In this case, which is the fully distributed approach, a scheduler is not used (Rusinkiewicz & Sheth 1995).

2.2 What is a task in e-mail?

E-mail is often used as a building block for groupware or CSCW, although e-mail was not originally designed for that purpose (cf. Section 1.3 on p. 7). In order to manage tasks, all workgroup members must often follow the same conventions. In the domain of e-mail, this may include everything from conventions that one should use the same type of headers in the message to agreements that one should employ the same type of language in the message. Standards facilitate the global acceptance of the e-mail system extended to accommodate for task management. One such standard for enhancing the task management capability of e-mail is Message Disposition Notifications (MDNs) (Fajman 1998). Receiving a simple receipt that the recipient has successfully opened a meeting request can impact an e-mail user’s list of tasks to do. We discuss MDNs and other types of standardised notifications in e-mail in Chapter 3.
How do we then define a task in e-mail? We see message threads as the key to identifying tasks. An e-mail message is never alone. For example, a message containing a meeting request with date, time, and place information, among other things, is sent to the meeting participants. The recipients reply with a yes or no message and, normally, also make clear to the original sender what the message is about, namely a reply to the previous meeting request message.

With regard to e-mail threads, quoting is used to preserve context in the dialogue represented in the threads (Severinsson Eklundh & Macdonald 1994). Furthermore, messages are often linked together by unique IDs via the e-mail protocol (Crocker 1982; Resnick 2001), at least according to the standard. For now, we just note that threading in e-mail supports context regeneration and the management of conversational history (Whittaker & Sidner 1996). In other words, a task in e-mail can be defined as a conversation taking place in a certain context that can be regenerated using the thread and information associated with the thread.

We have previously said that a task has a state. The state must be represented and tracked. In addition, in order to be able to manage tasks, information relating to pending tasks must be readily available. To summarise, we need to know how tasks are progressing, what is the task context, and if a user can employ the (e-mail) system to remind herself when a particular task has to be executed (cf. Malone 1983; Whittaker & Sidner 1996). We return to this topic in more detail in Chapter 3. We now move on to review some popular groupware products.

### 2.3 Groupware products

#### 2.3.1 INTRODUCTION

Groupware enables the members of a group to communicate, to collaborate (e.g., compose a document interactively), and to coordinate work (e.g., find a time to meet together). These are the three dimensions, or “the three C’s” of groupware infrastructure, as defined in 1995 by the Yankee Group (2001).

Our focus is on e-mail systems that integrate groupware functions so as to support the management of tasks. We divide the groupware functions into the following categories (Marshak 1999; Johansen 1988):

- **E-mail**: the basic functions of composing, sending, receiving, and managing (storing, retrieving, filtering, etc.) messages
- **Calendaring/Scheduling**: the maintenance of events and meetings in personal calendars and also group calendars, as well as functions for multi-user scheduling, *i.e.*, checking the availability of others’ schedules
• **Shared documents**: the ability to share information through shared or bulletin boards, public folders and the like, as well as the use of threaded discussions (conversation structuring)
• **Custom applications**: the possibility to use the system as a platform for customised applications, ranging from shared documents only to, for example, workflow (forms, routing, agents, etc.), data integration, and transactions.

Many of the groupware systems have remained in the research community, and have continued to evolve there. A few of the systems, such as INFORMATION LENS which evolved to OVAL (Malone, Lai, & Grant 1997), have been adapted to meet requirements of the commercial community. The integration of asynchronous and synchronous communication in cooperative work is discussed by, among others, Sakamoto and Kuwana (1993). This type of integration is beyond the scope of this thesis work.

Below we present a selection of products that have the aim of providing asynchronous distributed interaction support, and which build on existing e-mail services and database concepts. See also Terzis and Nixon (1999), and Hazemi and colleagues (1996b), for more exhaustive surveys.

In most of the cases the groupware functions have been integrated into a product that started out as an e-mail system. The products include all of the above categories of groupware functions.

### 2.3.2 Microsoft Outlook/Exchange

**Microsoft Exchange/Outlook** ("Microsoft Corporation" 2001) is a client/server-based messaging system. The **Microsoft Exchange** server runs on a Windows XP system and provides functionality to the Exchange client, named **Microsoft Outlook**.

**Microsoft Exchange/Outlook** is more focused on messaging than on task management. Nevertheless, **Outlook** provides some groupware functionality. **Exchange** includes capabilities for information-sharing (public folders) and calendar/scheduling functionality. The Exchange forms provide only a weak support for customised applications (Marshak 1999). The client, **Microsoft Outlook**, supports shared documents, forms, threaded discussions, and calendar/scheduling. It also offers basic personal information management (PIM) functions, i.e., the user can keep a calendar, add events and to-do items, schedule reminders, and jot down Post-it-like notes. Any information being written down must always be tagged to a particular person or a task, i.e., no random information is possible.
MANAGING TASKS ASYNCHRONOUSLY

MICROSOFT OUTLOOK is tightly integrated with the MICROSOFT OFFICE software package. This means that, e.g., MICROSOFT WORD can be used to check spelling and grammar, and documents can be routed for review from within MICROSOFT WORD, EXCEL, POWERPOINT, and other applications in the OFFICE environment.

Standards supported by MICROSOFT OUTLOOK include SMTP, POP3, IMAP4, and LDAP, in addition to MAPI (Messaging Application Programming Interface), which is MICROSOFT’s own architecture for messaging applications (e-mail, scheduling, PIMs, etc.).

2.3.3 NOVELL GROUPWISE

NOVELL GROUPWISE (“Novell” 2001) is a client/server-based groupware product from NOVELL and one of the pioneers of the groupware field. It combines document management; e-mail; group calendaring and scheduling; task management; imaging; and workflow in one integrated package.

The architecture of GROUPWISE is divided into domains, run by Message Transfer Agents (MTA), with Post Offices contained within, and each serviced by Post Office Agents (POA). A web-monitoring interface allows an administrator to connect a browser to a POA or MTA to collect statistics and change configurations. The agents are server-based processes performing the tasks of dealing with the message flow through the system as well as the critical task of providing full text and profile indexing services for documents in GROUPWISE. The documents are encrypted, compressed, and stored as BLOBs, i.e., Binary Large OBjects. GROUPWISE also has so-called Dynamic References, which are the automatic attaching of a referenced document to e-mail messages sent outside the GROUPWISE environment.

Approved users can search and browse another user’s calendar and make suggestions for meeting times. So-called search folders can be created and saved so as to locate frequently used documents and e-mail messages.

The GROUPWISE client is supported on most platforms (WINDOWS 98, 2000, and NT, as well as UNIX, LINUX and SOLARIS), but the server is only supported in WINDOWS NT and 2000. Both can run on NOVELL’s own platform NETWARE. A wide variety of protocols are supported (POP3, IMAP4, LDAP, and NOVELL’s NDS directory services), in addition to Secure Sockets layer (SSL) and Secure Multipurpose Internet Mail Extension (S/MIME).

2.3.4 LOTUS NOTES/DOMINO

LOTUS NOTES/DOMINO (“IBM/Lotus Software” 2001) is a client/server-based e-mail/groupware system from IBM/LOTUS SOFTWARE. The NOTES part refers to the client, and the DOMINO part refers to the server in the system.
The basic architecture of NOTES was developed before the appearance of web browsers. LOTUS NOTES/DOMINO is based on the concept of shared document databases where documents can be stored, viewed, updated, replicated (explained below), and routed as required. DOMINO is a collection of server processes, including database replication, e-mail routing, and indexing. There are two basic categories of databases. The first category is a \textit{shared database}, which resides on one or more servers, accessible to many users. Databases can be copied to additional servers for easier access. The databases are synchronised across the network through a process called \textit{replication}. The second category is a \textit{local database}, which resides on a workstation. Local databases are often personal databases, such as diaries or prototypes of new databases—or local replicas of remote databases.

NOTES/DOMINO is characterised by the \textit{ubiquitous access} to both messages and business data from any location, and via any device. Also, one common \textit{universal (unified) inbox} is used for all forms of messages (e-mail, fax, web pages, etc.).

In conclusion, NOTES/DOMINO can be scaled to any size of organisation. Its graphical user interface has a similar look and feel across all platforms (WINDOWS 95, 98, NT, 2000, and MACINTOSH). NOTES/DOMINO includes its own development environment for creating customised applications. All of the major Internet standards are supported, including security protocols such as SSL.

2.3.5 \textbf{MEMO}

MEMO from \textsc{Nexus} ("Nexus Secured Communications" 2001) began as a corporate-based e-mail system in the mid-1980s, originally developed by \textsc{VolvoData}. MEMO was previously marketed by \textsc{Verimation}. The IBM OS/390 operating system is today used as the server. The client is available for a number of platforms (WINDOWS 95, 98, NT, and 2000).

MEMO has calendar/scheduling functions, and document-sharing. Furthermore, it uses electronic forms, called MEMO forms, to manage tasks. A "workflow item" arrives in the MEMO mailbox just like ordinary e-mail messages, forms and calendar invitations. MEMO enables customised applications that can include workflow, data integration with legacy systems, and transaction integration via the intelligent forms capability and its published API.

MEMO is classified as a hybrid system, providing the advantages of client/server e-mail in functionality and the advantages of a host-based
MANAGING TASKS ASYNCHRONOUSLY

system in scalability, manageability, speed of upgrading system, and lower administrative cost (Marshak 1999). It also includes a JAVA ("java.sun.com" 2001) e-mail client. A new generation called MEMO OPEN CLIENT supports key Internet protocols, including POP3, IMAP4, and LDAP. The MEMO OPEN CLIENT is supported on WINDOWS 95, 98, NT, and 2000.

2.3.6 SUMMARY AND DISCUSSION OF GROUPWARE PRODUCTS

We summarise the features of the commercially available groupware systems presented in Table 2.1. A symbol “■” in the table means that the feature in question is available. A symbol “❑” in a column means that the system in question does not fully support the feature/function. The Personalisation and Customisation columns denote the user’s and the developer’s possibility, respectively, to modify and adapt the system. The Scalability column tells us whether the system in question is suitable for networked companies, not limited to a single intranet. Multi-platform denotes if the system runs on different computer platforms, such as for example WINDOWS NT, UNIX, LINUX, or MACINTOSH. We repeat that a task is a unit of work to be done. Task management is the ability of the system to allow the definition of these tasks and enable workgroup members and their systems to understand the task. Workflow, then, is the definition and coordination of the whole process of managing tasks, i.e., time and place for a task is added to the description of the group work.

When it comes to personalisation, MEMO has a special user database, which is deployed to store a user profile for each user. The profiles are key

<table>
<thead>
<tr>
<th>Product</th>
<th>Message-based</th>
<th>Personalisation</th>
<th>Customisation</th>
<th>Scalability</th>
<th>Multi-platform</th>
<th>Workflow support</th>
</tr>
</thead>
<tbody>
<tr>
<td>MICROSOFT OUTLOOK/EXCHANGE</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
</tr>
<tr>
<td>NOVELL GROUPWISE</td>
<td>■ ■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
</tr>
<tr>
<td>LOTUS NOTES/DOMINO</td>
<td>■ ■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
</tr>
<tr>
<td>MEMO</td>
<td>■ ■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
<td>■ ■ ■ ■</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of groupware products
elements in MEMO’s e-mail, calendar, forms, and bulletin board services. LOTUS NOTES and MICROSOFT OUTLOOK make use of so-called wizards to help and support the user configure and employ the system. NOTES applications are generally document-oriented, not transaction-oriented.

To fully enjoy the benefits of GROUPWISE as a groupware, it should be used as the messaging and e-mail platform as well. The same holds true for MICROSOFT EXCHANGE/OUTLOOK. Regarding customisation in GROUPWISE, third-party developers can create customised applications using an Object API, which is accessible from tools such as DELPHI, VISUAL BASIC, and C++. The API facilitates the applications to store and retrieve native GROUPWISE data or custom data fields.

MICROSOFT OUTLOOK/EXCHANGE has support for customisable applications—new functionality can be added to a site-wide installation by using the MICROSOFT OFFICE XP developer tools (WORKFLOW DESIGNER for EXCHANGE and SMART TAG SDK). In conjunction with the MICROSOFT WEB STORAGE SYSTEM SDK a workflow application can be created in MICROSOFT OUTLOOK/EXCHANGE.

MEMO OPEN CLIENT is mainly based on the widely used COM, which makes it possible to integrate MEMO with a large set of applications and components, technologies such as VISUAL BASIC, C++, JAVA as well as ACTIVEX components. A SYSTEM DEVELOPERS’ KIT (SDK) is also available for MEMO.

Regarding the scalability of LOTUS NOTES for Internet-wide information management, the replication function makes LOTUS NOTES unsuitable—a replication is the making of copies of databases over the network. However, LOTUS NOTES is available on many platforms, as noted above.

While IBM/LOTUS SOFTWARE (2001) has been focusing on knowledge management, MICROSOFT on the other hand has been focusing on messaging. Information-sharing is one way of handling tasks. As noted earlier LOTUS NOTES provides information-sharing through the use of shared databases, but includes an e-mail service for interpersonal communication. By using additional software a workflow can be implemented in a LOTUS NOTES/DOMINO system. An example of this is UNILINK DOCQMAN, which utilises the e-mail function, a graphical navigator, definition of access rights, and the grouping of users to provide an automatic workflow (see “UNILINK” 2001). MICROSOFT OUTLOOK/EXCHANGE makes use of public folders to share information.

MEMO makes use of forms (MEMO/FORMS) for structuring processes. The forms can be used for ordering, approval procedures, travel
requests, and reporting. By adding LIVELINK from OPEN TEXT to MEMO, the workflows can be designed and monitored graphically. LIVELINK comprises document management, full-text search engine, workflow, agents, and project collaboration tools.\(^3\)

## 2.4 Conclusions

Although end-user products differ tremendously—and also the messaging formats in some cases—they more or less manage to work together. This is thanks to the use of standards\(^4\) and “keeping the systems open.” Major industry trends today include the rise of the Internet, movement toward standards, and integration of groupware into base-level e-mail systems (Marshak 1999).

Concerning groupware functions, many end-user systems are proprietary. Such systems typically have benefits such as simple enterprise-wide calendars that can be maintained for every user, but only if the systems are of the same brand.

Companies that adopt groupware have often done so after years of experience with e-mail (Hazemi \textit{et al.} 1996a). Groupware itself has no value to a user unless others are using the same system too. The value of groupware increases with the number of users who are using it. As a part of groupware, e-mail provides a convenient way to exchange information and to transport documents, forms, memos, \textit{etc.} within a group of people. However, interacting and collaborating with other users requires that one also manages one’s own work and documents. The basic function of communication in e-mail—communicating asynchronously—and the continued employment of e-mail in a familiar, easy way have to be extended in a convenient way so as to support the management of tasks.

Much research about task support in office information systems was done in the 1970s and 1980s (see Croft & Lefkowitz 1984 for an overview). However, we argue that new developments have made the use of e-mail for managing tasks an interesting field of research. The brief survey of the commercial systems in this chapter has shown us the state-of-the-art of message-based systems in the business world. They all include an e-mailing function for interpersonal communication. The current (2001) trend is

---

3. The telecommunication company Ericsson was a large user of MEMO, but beginning in 1998 Ericsson replaced MEMO with OUTLOOK (Widell Örnung 1998). A negative side-effect from this is the increased danger for virus attacks in OUTLOOK.

4. Protocols and formats that are long-standing and currently much in use.
the universal (or unified) inbox, i.e., one place for storing and reading incoming e-mail messages, fax messages, voice mail, etc. This implies that there will be even more information to manage, and subsequently a support system is indeed needed. In other words, e-mail is a good starting point for implementing and introducing new functionality to a networked workgroup. Our goal is to support task management in e-mail. Our goal is not to support workflow, although the latter could be facilitated by the mechanisms that we aim to introduce.

We see workflow as the company-centred view of task management, i.e., workflow is a means to increase productivity, decrease costs, eliminate latency by automation, etc. These are all issues concerning the efficiency of work seen from a cost-based model of work.

The notion of task management that we use in this thesis is the user-centred view of handling tasks. This means that we concentrate on issues that are important for the individual user when managing tasks. Examples of these issues are easy access to task-relevant data, decrease of the risk for misunderstandings, and improvement of the overall efficiency when using e-mail to manage tasks.

New systems, such as Lotus Notes above, have emerged that are based on the principle of e-mail, but which also provide additional features that address the problems presented above. However, these systems are not widely adopted within organisations mainly due to the fact that effort is required to develop the applications to meet specific needs. That is, the “old” e-mail system cannot be used since the features only come with the new system. Also, most products are developed for the PC market, and the cost of software licenses is high (Hazemi et al. 1996b). However, the greatest problem is the fact that the new systems do not acknowledge and support the users’ ad hoc way of doing things in e-mail.

From a company’s viewpoint, introducing the full concept of workflow may increase efficiency. However, from an individual’s viewpoint this will probably also remove attractive properties of e-mail, making the acceptance of the new functionality difficult. Our focus is therefore on keeping the attractive properties of e-mail, but at the same time increasing the efficiency, as seen from the individual user’s viewpoint.

This ends the second chapter of the thesis. In the next chapter we discuss standards and examine the properties of Internet e-mail. Also, we investigate more concretely how e-mail can be used for managing tasks.
Chapter 3
Internet E-Mail

In this chapter we investigate the concept of e-mail from a number of angles. What are the characteristics of e-mail, and of Internet e-mail specifically? What is the technology used? How is e-mail used and what are the main problems involved with its use? We focus here on properties of e-mail that can be used for managing tasks. Among these properties are the simple sending, receiving, and forwarding of messages, as well as the grouping of messages into threads, and keeping track of replies. Closely related to message-based task management are the notions of classification and prioritisation, and structuring of messages, which we also explore. The chapter is concluded with a look at the e-mail user’s situation, what her needs for task management are, and how tasks can be represented in e-mail.

3.1 E-mail technology

3.1.1 INTRODUCTION
We have opted to focus on Internet-based e-mail and to investigate how to manage tasks in that specific environment. As discussed tentatively in the context of groupware products in Chapter 2, messaging on the Internet gives us an open, rich and widespread infrastructure for collaboration.

For a comprehensive survey of e-mail technology that covers both technical and user issues, see Palme (1995). Here we concentrate on the tech-
nology relevant for supporting task management in—and possibly extending an—e-mail system.

### 3.1.2 THE E-MAIL SYSTEM AND ITS MESSAGES

Originally, e-mail simply consisted of a simple transfer of a text file, with the convention that the first line of each message (file) contained the recipient’s address (Tanenbaum 1996). Today, e-mail messages can contain anything that is storable on a computer. The first Internet e-mail message with the now well-known @ (pronounced “at”) sign in the recipient’s address was sent in October 1971. The message was sent by Ray Tomlinson of Bolt, Beranek & Newman (BBN), the company that helped develop ARPAnet, the predecessor to the Internet (Seguller 1998, p. 106). He was also responsible for selecting the @ sign for this purpose—simply because “the person was @ this other computer.”

An e-mail system consists of two subsystems: the user agent (UA), which makes it possible for a user to read and send e-mail, and the message transfer agent (MTA), which transfers messages from the sender to the recipient (see Figure 3.1). In addition, a file system is used for storing messages. On the Internet the Simple Mail Transfer Protocol (SMTP), described in RFC 2821 (Klensin 2001; replaces RFC 821 in Postel 1982), is used for transferring the message from one MTA to another.

The message is encapsulated by an envelope described in the SMTP protocol. The envelope is used by the MTA and enables e-mail systems to

---

**Figure 3.1**: The Internet e-mail system model.
route the message. It also contains priority information, the destination address, and security level information, among other things.

The message itself consists of a list of headers, followed by a blank line, and a body. The information in the header is used by the UA and the MTA, and the information in the body is intended for the human user (see Figure 3.2 on p. 34). A header consists of a tag and a field of contents. An example is the To: header where the destination (the recipient’s) address is put, To: juhta@ida.liu.se. In this example, the UA expects an address of the form mailbox@location, which is the standard for Internet e-mail. Most of the headers are defined in RFC 2822 (Resnick 2001; replaces RFC 822 in Crocker 1982). The user is required to provide a recipient’s address (the contents of the To: header), and the message itself, of course. It is also customary to provide a subject (the contents of the Subject: header) to the message, a line that is used by the human recipient to identify the contents of the message.

Messages with content other than plain text, that is ASCII-coded, can be sent using the Internet standard named Multipurpose Internet Mail Extensions (MIME) (Freed & Borenstein 1996). MIME makes it possible to continue to use the RFC 822 format: it adds structure to the message body and defines encoding rules for non-ASCII messages so that MIME messages can be sent using the existing e-mail programs and protocols (cf. Tanenbaum 1996). In Figure 3.3 on p. 35 a message containing two attachments is shown. The first attachment is a file named hilbert, containing an application encoded in mac-binhex40, and the second attachment is simply the sender’s signature. The Mime-version: header tells the UA that this is a MIME message and the Content-type: header describes what type of MIME content it contains—in this case, multipart/mixed. The boundary attribute is used to delimit each MIME part. For each part a separate Content-type: header and an optional Content-disposition: header is given.

E-mail systems, from a user’s viewpoint, support some common basic functions, in addition to some more advanced functions. Tanenbaum (1996) talks about five basic functions: composing messages, transferring messages from sender to recipient, reporting to the sender about the message delivery,
CHAPTER 3

Example: Outgoing message

To: Neil Speirs <Neil.Speirs@newcastle.ac.uk>
From: N.F.Hall@ncl.ac.uk (N.F.Hall)
Subject: Re: Mail, News and M.Sc. Students
Cc:
Bcc:
X-Attachments:
Message-Id: <b06be60937021004e11f@[128.240.148.71]>

I could talk to them on Wednesday.

.. Nigel

Example: Incoming message

Received: from cheviot.ncl.ac.uk (cheviot.ncl.ac.uk [128.240.233.51]) by
burnmoor.ncl.ac.uk (8.7.6/8.6.10-cf revision 2 for
Solaris 2.x) with ESMTF
id MAA27224; Fri, 7 Nov 1997 12:28:13 GMT
Received: from ncl.callaly by cheviot.ncl.ac.uk id
<MAA19997@cheviot.ncl.ac.uk>
(8.7.6/ for ncl.ac.uk) with SMTP; Fri, 7 Nov 1997 12:28:13 GMT
From: Neil Speirs <Neil.Speirs@newcastle.ac.uk>
Message-Id: <AA03416.9711071230.callaly@uk.ac.newcas-
tle>
Subject: Mail, News and M.Sc. Students
To: N.F.Hall@newcastle.ac.uk (Nigel Hall)
Date: Fri, 7 Nov 1997 12:30:37 +0000 (GMT)
X-Mailer: ELM [version 2.4 PL24]
X-UIDL: 5b8ee0a2fca1b379ff2a39beb96f113b

The M.Sc. students would still like a talk on mail,
newsgroups and web use. I have lectures to them Mon 10-
11 T120, Wed 10-11 PG11 (Percy Building) Thurs 2-4 PG11
and Fri 10-12 Phy Lect Theatre 2. Can you (or a col-
league) talk to them for a while in any of these slots.
I would prefer Mon or Wed. so that I’m not harangued at
the Staff-Student meeting on Wednesday.

Neil

Figure 3.2: Typical headers and body of an outgoing (top) and an
incoming (bottom) Internet e-mail message.

displaying incoming messages, and disposing messages after receiving
them (see also Sproull 1991). The advanced functions include creating and
managing folders (labelled containers of messages) in the file system, as well
as sending carbon copies (Cc:), high-priority e-mail, and encrypted e-mail.
3.1.3 INTERNET E-MAIL STANDARDS

Rose and Strom (1998) define a “100% pure Internet” product as one which faithfully implements Internet standards and protocols as a default. A related concept is the open system. This is defined as a system where the protocols used are standardised and freely available, and in which there exists support for clients and servers running on more than one type of operating system. The emergence of the Internet has shifted the focus to standards, and standards provide scalability.
Companies are generally trying to break away from proprietary LAN-based e-mail systems to open, Internet-based systems that can integrate both internal and external communications (Widell Örnung 1998; Rose & Strom 1998; Marshak 1999). This is also evident from the brief survey of groupware products in Chapter 2.

Standards make it possible to integrate software. Internet standards foster many benefits, including client/server interoperability, high system scalability, and seamless communication across the Internet. The key Internet standards include, among others (see also Kurose & Ross 2001):

**SMTP**  
*Simple Mail Transfer Protocol* (Resnick 2001) is the basic protocol for sending and receiving textual e-mail.

**MIME**  
*Multipurpose Internet Mail Extensions* (Freed & Borenstein 1996) makes it possible to send various types of content (textual and nontextual) with SMTP, and not just 7-bit ASCII.

**IMAP4**  
*Internet Message Access Protocol 4* is a method of accessing electronic mail or bulletin board messages that are kept on a (possibly shared) mail server (“The IMAP Connection” 2001), without the need to transfer messages or files back and forth between these computers.

**POP3**  
*Post Office Protocol 3* is an older mail access protocol, designed to support off-line message access, wherein messages are downloaded and then deleted from the mail server (Myers & Rose 1996).

**LDAP**  
*Lightweight Directory Access Protocol* enables clients to access information in LDAP-based address directories—also, directory information can easily be synchronised with any other LDAP-compliant directory.

Furthermore, standards for the web, such as HTTP (HyperText Transfer Protocol), HTML (HyperText Markup Language), XML (eXtensible Markup Language; see Bradley 2000 and Cover 2001) and JAVA (“java.sun.com” 2001), are being supported in e-mail environments, in addition to security-related standards. Examples of the latter are PGP (Pretty Good Privacy) and SSL (Secure Sockets Layer) (cf. Kurose & Ross 2001). The real advantages of standards-based e-mail are the ability to mix and match products from multiple vendors and the promise of no vendor lock-in (Marshak 1999).
3.2 An exploratory case study of e-mail usage

3.2.1 Introduction

Do we really need special support for task management in e-mail? Doesn’t e-mail work well just as it is? How do people try to manage tasks in e-mail? How and why are they using the currently available functions in their e-mail clients in this process?

These were the main research questions we wanted to investigate. Since information technology is only one part of an information system, we wished to get an understanding of the general workspace a user has for performing message-based tasks. We wanted to verify the diversity of e-mail usage (Mackay 1988; Takkinen 1994) and also get an overall picture of how much (or how little) certain functions of the e-mail clients really are used. Especially interesting functions were those associated with task management, such as forwarding, filing, structuring, etc. of messages, and the use of aliases and the address book.

By exploratory case study we mean an empirical investigation where many variables are studied at the same time, and where control of the conditions is limited (Yin 1994). There is no requirement on control over behavioural events in a case study, and the study focuses primarily on contemporary events.

3.2.2 Method

We designed the exploratory case study with the purpose of investigating how and why a user manages tasks in e-mail, i.e., what kind of actions (different functions and commands) and tasks a user performs, related to the roles a user has. The study also included a simple survey of the types of e-mail clients used. We adopted a user-centered approach inspired by the ideas presented by Allen (1996). Both interviews and a questionnaire were employed. During interviews, the subjects were asked to refer to actual messages and exchanges of messages in their e-mail.

The questionnaire, which was originally in Swedish but later translated into English (see Takkinen 2000), was used primarily to collect data about the scale and patterns of usage of e-mail systems. We wanted to get a picture of the scale and patterns of usage of functions such as forwarding, filtering, templates, aliases, and the address book. Interviews were used to probe knowledge and the use of features of existing e-mail systems, as well as other (electronically or not) available sources, to manage tasks.
The questionnaires were distributed by e-mail to mailing lists within two computer-related departments at the university, to two student choirs, and to a group of alumni computer science students. A total of 158 questionnaires were returned, which we grouped into four categories: doctoral students (55 in all), other academic people working at the university (54), industry people working outside the university (32), and undergraduate students (17). The groups are henceforth coded DS (doctoral students), AP (academic people), IP (industry people), and US (undergraduate students), respectively. We concentrated on (current and future) scenarios where tasks were delegated and information was disseminated. Furthermore, four subjects from the university were interviewed: a helpdesk operator, a doctoral student, a director of a research group, and an administrator.

3.2.3 Results
The main results of the study were (Takkinen & Shahmehri 1998b):

• **advanced formatting** of messages by using extra bold type, HTML, etc. is rarely done because it takes time, because the messages are mostly short, and also because the receiving side cannot be presupposed to see the same layout; instead, documents are created in a word processor and attached to the message
• **templates** (other than signature files) are not widely employed because they are a hindrance and also because most messages are short
• **forwarding** messages is very popular: a new message is created via a forward button, the recipient(s) *(To: and Cc: headers)* and a *Subject:* header are added to the message, and some explanatory text is put in the body of the message; it is also notable that the forwarded messages tend to be of lesser importance to the person who is doing the forwarding
• **confirmation** of the delivery of messages is generally cumbersome with Internet e-mail and typically the telephone is used to confirm important messages; some subjects in the study also used ICQ (“ICQ” 2001) or the UNIX command *finger* to verify message deliveries
• **mobility**; accessing one’s e-mail is often done from different computers using different e-mail systems, and acting in different *roles* depending on the place where the e-mail is accessed from.

Furthermore, aliases and the address book are used heavily. The DS and the IP tended to read e-mail as it arrives, while other AP read mail in intervals. Generally, e-mail is regarded as extremely useful, especially by the
INTERNET E-MAIL

Filtering is used mostly by the DS (41%) and the IP (38%) in the study. The AP had the largest spread of the number of mailing lists, 1–50, while the other groups had a more modest spread of 1–22 (DS) and 1–15 (US and IP). The US were most unsure (24%) about the number of mailing lists they were subscribed to, as compared to the other groups (7–15%).

3.2.4 SOURCES OF ERROR IN THE METHOD

Three questions in the questionnaire (Takkinen 2000) were badly designed, which caused some problems for the subjects when trying to answer them:

- In question no. 6 the term “signalling function” was not understood by many—it was intended to be interpreted as the sound used by the e-mail system when a new message arrives. Instead it could be interpreted as any type of signal, including visual.
- In question no. 10 the term used for “folders” in Swedish (“mappar”) was not well-known to some subject.
- In question no. 19b the phrase “In what order are the messages in the list displayed?” in Swedish could be interpreted in two different ways: “How is your system currently configured to display it?” (the correct interpretation) or “Which alternative ways do you have available for displaying the list?” (the incorrect interpretation).

The answers to these particular questions were not used in the final analysis of the results.

3.2.5 DISCUSSION AND CONCLUSION

If we take a look at the most commonly available applications for reading and handling e-mail, they can, at first glance, provide a user with much value. For example, the average e-mail client provides basic functionality to manage messages as follows (Takkinen & Shahmehri 1998a): filtering (classifying and prioritising) messages based on keywords located in message headers is common, as is the use of folders to store messages, and two-paned and three-paned display of folders and messages on the screen. The standard features to handle a message include the ability to compose, reply to, copy (Cc:), and blind copy (Bcc:) messages to other users. Furthermore, messages can typically be viewed by sender, date, or size. Also, the e-mail client usually has a text search capability: the personal archive of e-mail messages can be full text indexed, so that keyword or Boolean queries are possible (Whittaker & Sidner 1996).
Formatting It was a bit surprising that advanced formatting of messages was not so popular to use. In an earlier study (Takkinen 1994) the most requested function in e-mail was better functions for layout in messages. One of the explanations for this difference is probably the good handling of attachments that has evolved since the introduction of MIME (Freed & Borenstein 1996). Today, if text with better layout is wanted it is simply created with a word processor and sent as an attachment (cf. Section 3.1.2 on p. 32). This was evident from the answers in the questionnaire in our study.

On the receiving side the different characteristics of e-mail messages can be suppressed or reinforced in the representation on the screen.

Templates E-mail messages are by default already to some extent semi-structured (Malone et al. 1987a), with respect to their structure, layout, and content. The users’ categorisation of messages in folders (labelled containers of messages) by the contents of the Subject: header is, according to Malone and colleagues (1987a, p. 116), indicative of that templates should be natural to introduce in e-mail. Templates are good for constructing formal messages, such as meeting announcements, bug reports and ordering of goods. The basic idea is that templates can be used for at least partially formatting the information passing through an organisation, and that doing this allows one to build a flexible information dissemination system. Moreover, these partially-structured messages can serve as the data structures for secondary systems such as calendar management (Malone et al. 1987a). In other words, if the messages are made more structured they can be more automatically processed—or forwarded from one user to another—according to each user’s role in the group. If a scripting language is added to this kind of system, workflows can be described and implemented (Hazemi et al. 1996a).

Although templates would give e-mail messages more inherent structure, they would also give less freedom to the e-mail user. The mandatory use of templates when composing messages should be limited, or the needed message structure should be supplied in a more automatic way. The user does not want to spend too much time selecting items from menus before being able to compose and send a message.

Confirmation Internet e-mail users have a common problem with how to confirm the delivery of a message and to be notified when a message has been opened (and subsequently read) by a recipient. This is a problem associated with the open architecture of Internet e-mail. Confirmation of the
INTERNET E-MAIL

delivery, opening, or reading of messages are all functions that are important to consider when managing tasks. Notifying a sender that a message has been delivered to the recipient can be solved by the use of the Delivery Service Notification (DSN) Internet standard (Moore 1996) (see Figure 3.4). Basically, DSNs define a special MIME type to report message status information. The types of information that are transmitted are successful delivery, delayed delivery, or failure, with possibly additional diagnostic information. A DSN is not able to report if a user has actually looked at a message, or any other status beyond “it was placed in their mailbox.”

Figure 3.4: An example of a Delivery Status Notification (DSN).

The problem of notifying a sender when a recipient has opened a message can be partly solved by employing the Internet standard for Message Disposition Notifications (MDNs), which are put into the message header (Fajman 1998) (see Figure 3.5 on p. 42). MDNs build on the DSN framework (see above) to report the final disposition (seen, discarded, etc.) once it has been placed in a user’s mailbox. Receipt of a “read” MDN does not imply that the message was understood or dealt with—merely that it was displayed for the user to read. The user may have been asleep, on the phone, or just did not want to act on your mail. Not all mail programs support MDNs. If the recipient’s software does not support MDNs, the sender will not get one back. Other examples of notifications, in addition to DSNs and MDNs, are (Palme 1998):

- non-delivery status notification: the message could not be put into the mailbox of the recipient
- subscription notification: a new member has subscribed to a mailing list or a member has signed off the list

Date: Fri, 27 Jul 2001 11:57:27 +0200
From: Mailer-Daemon@ida.liu.se
To: juhta@ida.liu.se
Subject: Message status – delivered

1. The message that you sent was delivered to the following:
   patla

2. Information about your message:
   Subject: Re: New version of report
   Message-Id: <200107271139.NAA11632@obel10.ida.liu.se>
• request-for-membership notification: a person wants to join a closed mailing list, for which you are the moderator, and you are asked to either accept or reject the request.

As noted above, the telephone is often used to confirm the delivery of messages. An example of an open, Internet-based e-mail system that incorporates synchronous communication into the asynchronous environment of e-mail is given by Mantel’s work (1999). Here, an extra MIME type is defined to accommodate for the function of automatically switching to a synchronous communication form. In a prototype the UNIX VTALK program is used for the synchronous connection. However, the asynchr-
INTERNET E-MAIL

nous nature of e-mail—that a sender and a recipient can send and read messages at different times—is one of the main reasons for the success of e-mail, so care should be taken not to disrupt the balance.

**Mobility** The mobility of users suggests that different classification and prioritisation schemes of messages may be needed, depending on the different electronic (user) environments and user roles. Users access their accounts on systems from vendors such as MICROSOFT and SUN via POP3 and IMAP4 e-mail clients, or HTTP browsers. The same user can be using different e-mail clients at different times, depending on where she is and what equipment (computer and software) is available.

### 3.3 Patterns of e-mail usage

Typically, an e-mail message is associated with a sender of some sort, usually a person. Furthermore, each message normally deals with one subject or topic, which often is stated in the title, *i.e.*, the **Subject**: header in the message. E-mail messages can be studied with respect to their structure, layout, or content. The differences between these three aspects can be clarified in the following way:

- **The structure** is defined by the protocol and consists of headers (for example, **From**: and **Subject**: headers) and a body (text and MIME parts) (Resnick 2001; Freed & Borenstein 1996). These parts correspond to the structure found in books: table of contents, paragraphs, lists, headers, *etc.*
- **The layout** is created by the user and is often governed by the functions available in (and handled by) the e-mail client (the e-mail reader software). Examples of these functions are the definition of the type and size of available fonts and the placement of text (center, right, or left).
- **The content** is supplied by the user and is often governed by the content of a previous message. The content is characterised by quoted material (Severinson Eklundh & Macdonald 1994), and language (style and type), among other things (Yates & Orlikowski 1993).

With regard to the construction of the **Subject**: header, users on mailing lists (e-mail-based discussion groups) tend to have certain conventions. A typical example is the adding of a prefix to the **Subject**: header of a message to simply indicate the mailing list’s name. Mailing list software often
supports this. For example Yahoo! Groups (2001), a free e-mail group service, supports the automatic addition of a prefix (by default the mailing list name) to each message sent to the mailing list.

**The inbox and other folders** Many users use the inbox, where incoming unread messages are delivered, as a *to-do list*. That is, the inbox is used as a list to remind the user of things that have to be done: incomplete tasks, unfiled information, and ongoing conversations. The inbox is usually the first thing displayed when starting the e-mail client, which means that the to-do list is easily available. E-mail messages in the inbox can be an important determinant of how people will spend their working day (Whittaker & Sidner 1996). There are users who also send separate messages addressed to themselves, containing essential information for some future task to be done—all in order to get the information into the e-mail system and subsequently stored in the inbox/to-do list.

Some users create a separate folder for storing to-do items. However, this may be dangerous since it requires the user to remember to browse the special folder on a regular basis (Lehnert 1998). Bälter (1998) has concluded that as long as the number of messages is less than a few hundred, users seem to be able to cope with their e-mail messages using only the inbox and no folders (if we do not treat the inbox as a folder). Zero to three folders seem less efficient than four to approximately twenty folders for searching manually—thirty or more folders is not efficient. Bälter also studied the relationship between the time spent archiving messages and retrieving them, and noted that it is important to improve the facilities for archiving messages for those that search manually—and to improve the search tools for those that use search tools.

**Gathering information for decision-making** Because people do other things between answering e-mail, tasks which require many interactions will take longer through e-mail (see Palme 1999b). A typical process is the decision-making when trying to schedule a meeting. The usual algorithm (someone proposes a date, everyone checks if the date is suitable, if not then propose a new date, etc.) when deciding face-to-face will not work. Palme (1999b) suggests the following algorithm for scheduling a meeting in e-mail:

1. The initiating person proposes five alternative dates, and asks everyone to reply to that person personally, indicating which of the dates are acceptable to them.
2. The initiating person then collects the responses, checks which dates
suit the largest number of people, and schedules the meeting for that date.

3. If none of the suggested dates are good enough, go back to step 1.

**Filing and filtering** Users have developed both automatic and manual strategies for handling messages and the tasks related to them in e-mail. Often messages are manually collected, sorted and prioritised. A typical manual strategy is to collect all messages in the inbox, read them, and then move the messages to different folders (Whittaker & Sidner 1996). The classification can be partly automated by writing filtering rules that sort the messages into the folders. The rules are typically *IF–THEN rules* that trigger on keywords in the message headers (*To:*, *From:*, etc.). In addition to these segments of a message, other parts of a message can also be used for classification purposes (Takkinen 1997): its *overall structure* (the whole message) and *references to other messages*. We have previously examined users’ classification of messages (Takkinen 1997) and the usability of a user interface for defining and using filtering rules (Takkinen 1994).

Classification and prioritisation play a central role in the management of tasks—the two methods interact to make a system usable to a user. For example, e-mail messages can be classified by who sent them and then sorted according to prioritisation order to get an ordering of the tasks that the messages represent. The prioritisation order depends on the importance of the tasks involved. For example, if getting a message from the supervisor is more important than an update to a weekly meeting schedule, then the first message should appear first in a list of pending tasks.

With many tasks to handle, a user needs to be supported by a scheme or system that can help her classify her tasks. Finding an update to a meeting schedule, getting the response to a message about a submission one has sent to a conference chair, or locating a message from someone that one has tried to reach earlier the same day can be crucial to staying in touch and on schedule. When the tasks have been classified and prioritised, the user can start acting on the tasks, delegating some of them, and so forth.

We know from previous work (Mackay 1988; Whittaker & Sidner 1996) that e-mail today is used for diverse purposes, among them information and task management. These purposes are extensions to the basic function of e-mail, which is message-based asynchronous communication. Since e-mail is used for other purposes than what it was intended for, users experience *information overload*, which subsequently leads to *e-mail overload* (see Chapter 1). It is evident that the user needs to be able to view informa-
tion represented in different ways, according to different information needs (see, for example, Takkinen 1997).

The viewing of information according to threads, which represent tasks, needs mechanisms in the e-mail client to associate incoming and previously stored messages with threads—and to handle the threads in different ways. Next we investigate the task as related to the concept of threads.

3.4 The thread as a task in e-mail

Task management requires at least the following functions in an e-mail client (the list is modified from Whittaker & Sidner 1996):

- **conversational threading** and **semantic clustering** of other messages that normally do not “cling” together via headers—both of these functions make reminding about tasks and tracking tasks possible since the number of items displayed is reduced
- **marking** of particular messages as requiring action
- **programming of reminders**, for actions that cannot be done immediately.

The last point is an interesting extension to the use of reminders that we discussed above. In that discussion we also noted the use of the inbox to list messages that require action. The second point in the list above is an alternative way of marking a message as requiring action, particularly useful if the user is storing messages in separate folders and not using the inbox as a to-do list. We examine the first point, conversational threading, more thoroughly in the rest of this section.

As we have noted already in Chapter 2, a task can be defined as a conversation. A conversation in e-mail is an exchange of more or less naturally chained messages, *i.e.*, a message is often a response to a previous message. The **parent** of a response is the message that it responds to. The parent may be a response to another message, and so on. Other messages can also have the same parent. We call a collection of messages with a common ancestor a **thread**. The initial message in a thread is called the **root**. Small tasks can be contained within these threads.

Using the language of the Internet standards (Resnick 2001; Horton & Adams 1987), a thread is a series of messages that are linked via the **In-reply-to**: or **References**: headers to a parent message (or several, in

---

6. This is supported mainly by the use of the information in the **In-reply-to**: and **References**: headers in the mail protocol.
the case of the \texttt{References} header (see Figure 3.6). The figure shows a root message A with one reply message B. Two other messages C and D reference message A, not necessarily being replies to A. Message B has one reply message E, and E has one reply and one reference. Message C, lastly, has one reply message F.

There also exists a \texttt{Supersedes} header (not shown in the figure), mainly used in net news (some net news clients use \texttt{Replaces}). This header refers from a new message to an old message and cancels the old message (Lindsey 2001; Palme 1999a).

Typically, the reply command copies the \texttt{Message-id} header contents to the \texttt{In-reply-to} header of the child. Also, the value contained in the \texttt{Subject} header is copied to the corresponding header in the response, with a \texttt{Re} token in front of the value.

For a message to be linked into a thread it must contain a \texttt{Message-id} header. The \texttt{Message-id} gives the message a unique identifier, an ID (identification value). In order to conform to RFC 2822 (Resnick 2001), the ID must have the format:

\texttt{<unique@full\_domain\_name>}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{thread_diagram.png}
\caption{An example of a thread in e-mail.}
\end{figure}
Here, the full_domain_name is the full name of the host at which the message entered the network, including a domain that the host is in, and unique is any string of printable ASCII characters, not including < (left angle bracket), > (right angle bracket), or @ (at sign). Here are two examples:

<200003022112.WAA10736@obell.ida.liu.se>
<"16913 Fri Mar 3 13:05:59 2000"@bnr.ca>

The address encoded in the ID is not usually an address that can receive e-mail. The ID is in theory tokenisable, i.e., the following three types of tokens are allowed:

- punctuation: <, >, comma, semicolon, and colon
- address symbols: at sign (@), dot
- encoded strings: atom, domain literal, quoted string.

An atom is a string of one or more characters terminated by the end of the field value or by any of these characters: space, tab, @, <, >, [ left parenthesis, comma, semicolon, colon, dot, or double quote. However, in real life it is not recommended to use quoted strings, spaces, tabs, or comments inside message IDs (as in the second ID example above). Any string starting with [ and ending with ] can be encoded as a domain literal.

The common ancestor or initial message is called the root message. For example, the References: header contains the message IDs in sequential order, with the last ID identifying the parent. This list is truncated if it becomes too long, but the ID for the root message in a thread is always preserved, even where truncation has occurred (Resnick 2001).

Here we pause to make an observation regarding the In-reply-to: and References: headers. The message IDs in them are, at least in theory, globally unique. However, with regard to the construction of the message ID things look different. As observed by, among others, Zawinski (2001) the message ID in the In-reply-to: header is really free text since the syntax defined in RFC 2822 allows it to contain basically any text at all. Most of the time, however, the message ID contains something useful. The References: header, on the other hand, follows the RFC 1036 (Horton & Adams 1987) definition for message IDs used in net news (USENET news). This definition is tighter than the definition in RFC 822. Consequently, RFC 2822 the successor to RFC 822, mandates the use of the message ID definition used for the References: header.

Some e-mail clients allow threaded display of messages, though this is less common than in net news readers. For instance, the EXMH e-mail cli-
INTERNET E-MAIL

ent (Welch 2001) allows grouping of messages according to several criteria: by the same sender, by date, or by the same subject. Archives of e-mail can be threaded using HYPERMAIL (“Hypermail Development Center” 2001; “Hypermail Home Page” 2001), which creates HTML pages with links between messages. The algorithm used by HYPERMAIL is as follows: it first tries to find an ID in the \texttt{In-reply-to:} header and match it to any known message. If this fails, HYPERMAIL looks for a matching date string in the \texttt{In-reply-to:} header, and as a last resort it tries to match on the \texttt{Subject:} header contents (after removing any \texttt{Re:} tokens). However, as discussed above, this algorithm will fail. For a more robust threading algorithm, implemented in JAVA, see Zawinski (2001). Berthold and colleagues (1997) have investigated the use of an auto-associative neural network (ANN) to classify “good” and “bad” messages in the sense of their participation in a thread in net news discussions. The messages were stripped of identification codes (message IDs and \texttt{From:} headers, among other things). A “good” message is then a message which is at least referenced (in free text) by one other message, while a “bad” message is not referenced at all.

All of the structural information described above, inserted by the UA (the e-mail client) and/or the MTA, is useful for recovering threads. However, many UAs truncate the \texttt{References:} header because of space limits. Also, the use of \texttt{In-reply-to:} is actually optional (Resnick 2001), and the format and nature of the ID is only loosely constrained, as exemplified above. Moreover, the same \texttt{Subject:} can occur in two different threads out of pure coincidence. Also, a user can create a message that is actually a reply to an earlier message but without using the reply function of the e-mail client. This destroys the Internet e-mail thread described above. More common is the case when a user takes an old message and employs the reply function simply to avoid writing the address of an intended recipient. The address is automatically inserted into the \texttt{To:} header, but the old \texttt{Subject:} header is kept. Bälter (1998), among others, has reported on this user behaviour, which is a sign that the address book function used together with e-mail clients often lacks functionality. Apart from this creative user behaviour, there are also inconsistencies between e-mail clients which can break the structural information needed for threading. Threading systems based on structural information are thus only partially successful and this situation may be unlikely to change.

All sorts of communication must be able to handle exceptions. In workflow systems, good exception handling is crucial. When a thread breaks
down in e-mail, a backup system must be used. Lewis and Knowles (1997) have studied machine learning (ML) support for threading in e-mail and also the appropriateness of ML for exception handling. They propose that threading must be treated as a language processing task. For example, the root message most likely will have little or no quoted material in it, while other messages referring to the root message do contain quoted material. This information can be used to group related messages together, although the messages do not have the correct (or any) information in their References: and/or In-reply-to: headers.

Despite these inadequacies in current threading functionality, people are still able to follow discussions. This suggests that the messages indeed contain contextual information which is needed to understand their place in an ongoing conversation (cf. Lewis & Knowles 1997).

3.5 Summary

E-mail has changed the way people operate (Sproull & Kiesler 1991)—and it is usable by virtually every person who owns or uses a computer. The keys to this success have been the adoption of TCP/IP as the standard transport mechanism and the adoption of the (original) RFC 821 and RFC 822 standards. These protocols enabled e-mail to be transferred between different systems and different e-mail clients, and have now (2001) been updated for the current Internet.

We have described the domain of Internet e-mail. We concentrated especially on the aspects relevant to supporting task management in this domain. We examined the e-mail thread, its current capability to represent a task in e-mail, and some suggestions for making it more robust. Threads collect the messages into a common context (discussion topic) and are therefore interesting as a modelling tool.

In addition to threads, we also discussed the use of to-do lists, which list messages that require action. These are the pending tasks in the e-mail system. Reminders are related to the to-do list—they are either passive, such as the simple browsing of the to-do list at sporadic intervals, or active, such as the programmable reminders suggested by Whittaker and Sidner (1996). We note that browsing the to-do list is often sufficient as a reminder of tasks to do. So, by combining the to-do list, the reminders, and the threads, we have access to relevant status information about a task.

This ends the first part of the thesis. In Part II we develop a conceptual model for supporting task management in e-mail. A language is also described that supports the model. We also present related work.
Chapter 4
A Conceptual Model for Task Management

Here we present a conceptual model for supporting task management in e-mail. We introduce the concepts that are central to our reasoning about the problem. We also explain our reasoning in choosing this modelling approach. We give an overview of the model and subsequently describe each of the components of the model. We also relate the model to the notion of organisational memory presented earlier (see Chapter 1). To show the functionality of the model we conclude with the scheduling-a-meeting scenario, using two specific views: one user-centred and one task-centred.

4.1 Introduction
In Chapter 1 we noted the fact that e-mail has evolved to be more than just an asynchronous communication medium. Over time features have been added to accommodate other functionality, resulting in e-mail overload. Users have developed more or less cumbersome and ad hoc solutions to handle tasks in e-mail.

To recapitulate, some of the task-related areas that we have found problems in (cf. Chapter 1 and Chapter 3) are as follows:

- preserving task context, for example, having information relating to a current task readily available
- switching between task contexts
- getting an overview of one or more tasks and related information
• being reminded about tasks
• mixing formal and non-formal communication and still keeping it within a task context
• mixing electronic and manual tasks
• keeping the positive characteristics to Internet e-mail intact: asynchronous, an open architecture, well-known, simple, etc. (see Chapter 3).

We conclude that a restructuring of the application domain of e-mail, as well as the development of a conceptual model for task management in e-mail, are needed.

4.2 Selecting a modelling approach

The different types of a user’s tasks were discussed in Chapter 1 and are shown in Figure 4.1 on p. 55. The placement of the note pad and the at sign (@) in the cube represent the types of tasks that we are focusing on modeling, although the other types of tasks are also included in our work.

A model can be either developed from scratch or based on an existing model. With this in mind we find that there are three different approaches for developing a model:

1. Define a general model for task management: In this approach a general model for task management is defined from scratch, and not necessarily and specifically for e-mail.

2. Define a model based on an existing workflow model: Here, an existing workflow model is used as a starting point and then e-mail is incorporated into it. The model is then modified accordingly.

3. Define a user-centred model, with e-mail included from the start: Another way to develop a model is to employ a user-centered approach. Here, the activities (typically unstructured and non-formalised tasks) of the e-mail user are studied. The activities give clues to the functionality that has to be included in a model for the management of tasks in e-mail.

There are both advantages and disadvantages to each of these approaches. The two first approaches are task-centred approaches, although with different emphasis, while the third one is user-centred.

In the first approach the tasks are defined beforehand. A model is thereafter developed for handling these tasks. Since the model is general, both electronic and manual tasks (see Section 1.2 on p. 4) can be included in it. Also, because the model is developed from a number of predefined tasks,
The tasks can probably be handled efficiently in the model. However, in this modelling approach the user is included in the model in a later phase, if ever. This means that the user is made more anonymous than the task in the model.

In the second approach the coordination of the tasks is included in the model from the beginning. The workflow model presupposes activities that involve the coordinated execution of multiple tasks by different processing entities (cf. Chapter 2). The tasks are typically structured and formalised from start—business processes are the typical example of these types of tasks. Although there are many companies that implement their own workflow models to automate processes, a special Work Flow Management Coalition (WFMC) (2001) exists for standardising workflow and enabling different workflow management products to work together. The use of standards makes it possible to implement a model on different platforms and also to integrate workflow with, e.g., e-mail. However, from an e-mail-based task management viewpoint, there are problems. Including e-mail into an existing workflow model means either relaxing the requirements for the tasks in the model or restricting the use of e-mail. That is,
either the workflow model becomes less formal, or the use of e-mail is made more restricted.

In the third approach, the user-centred one, the tasks can be allowed to initially be non-formalised and unstructured. In other words, we do not require the tasks to be formalised and defined from scratch. By exploring what the user does, functionality needed for defining and handling tasks can be discovered and a model developed. Since the application domain (in our case, e-mail) is studied explicitly in this approach—along with the user’s activities in the domain—the application as well as the user are both included in the model from scratch. There is, however, a trade-off needed between how much of the user is included in the model and how much functionality is introduced to the user. The model can become overly user-centred or introduce too much functionality—chaos, instead of flexibility will be the result. An advantage of the third modelling approach is that it is simple and does not presuppose definition of tasks (as in the first and second approaches above) or formalisation of tasks (as in mainly the second approach). The third modelling approach includes the individual user in the model, which means that communicating the model to (both specific and general) e-mail users will be easier to do. Consequently, compared to the first approach described above, the third approach makes the user less anonymous in the model. For these reasons we select the third approach, i.e., the user-centred approach.

4.3 The scheduling-a-meeting scenario

Before we present the conceptual model in detail we present the scheduling-a-meeting scenario. We use the scenario to exemplify the use of our conceptual model. Subsequently in the thesis, this scenario is also used to exemplify the architecture and implementation of the model.

A second, more advanced scenario, the making-conference-arrangements scenario, is described in Appendix A.

In any organisation, the ability to manage appointments or meetings is required when events are to be scheduled. Figure 4.2 on p. 57 shows the state diagram for scheduling a meeting. An initiator of the meeting composes (state b) and distributes (state c) a request for an appointment, including one or more possible dates and times for the meeting, the duration, a place, and a list of participants. Documents, possibly including a meeting agenda with comments, are also distributed at the same time. The request sent by the initiator usually requires the receivers to respond—the
initiator waits for responses in state $d$ in the figure. The response can be either negative (a simple "no," sometimes with an explanation) or positive ("yes"). The response can in some cases even be "maybe." Moreover, there is a particular time (deadline) by which the responses have to be made.

With e-mail, there is always the possibility for delivery error (transition from state $d$ to $c$ in the figure), which can be caused by, e.g., an incorrectly written receiver’s address. The timeout in the same transition from state $d$ to $c$ is the result of the initiator using different types of reminders to remind herself to check the task status. In this case, a typical task status is that responses from one or more of the receivers are missing.

When the initiator checks the status of the task she can decide if the meeting should be scheduled or not (states $e$ and $f$). In the latter case the initiator probably needs more information from some of the receivers. She can select either to resend the request for a meeting in order to trigger a response, contact the person(s) in question by other means—typically by telephone—or to schedule the meeting anyway. The task status is discussed more in the subsequent section, in conjunction with the detailed description of the conceptual model.

**Figure 4.2:** A general state diagram for scheduling a meeting in a message-based system.
4.4 The conceptual model

4.4.1 AN OVERVIEW

The conceptual model for managing tasks in e-mail is shown in Figure 4.3 on p. 59. The model presupposes an existing e-mail system. The basic unit handled in the model is the message. Messages are grouped together into conversations. In other words, conversations are exchanges of messages. Conversations are used to represent tasks in the model. Before we continue to discuss the concept of a conversation and its relation to a task’s status and context, we describe the other components in the model.

There is a two-stage classification of messages in the model, consisting of sorting and prioritisation (see the small circles in the upper half of the figure). In the first stage incoming messages (see the upper left half of the figure) are classified as either being part of a current conversation or not. Current means that the user wants the conversation to be logged, which in turn means that the messages belonging together should also be grouped together automatically by the system. In the second stage of classification the messages that have been classified as belonging to a conversation are sorted into the correct conversation. Furthermore, all the current conversations are sorted or reorganised according to a prioritisation order. The other messages are sorted into groups (folders) that have been predefined by the user.

The classification in both stages is based on information available in the electronic environment of the user, represented by the large shaded area in Figure 4.3. The electronic environment consists partly of personal sources and partly of external sources, available electronically to the user. Some of the information sources are shown in the figure: one external source (for example, a database of employees) and a number of personal sources (the calendar, the rules database, and part of the personal database of the user, i.e., the groupings of messages in conversations and other messages). The outgoing messages are also part of the electronic environment.

Information from the electronic environment is also used for prioritising the messages that end up in the conversations group. Typically, the prioritisation information comes from the user’s calendar of events and recently sent messages. For example, a message about a particular business contract might be quite urgent before an approaching contract deadline, and therefore should be placed higher up in a prioritised list. However, after the deadline this message is not necessarily urgent.

The electronic environment contains “traces” of the interactions that the user has with people (interaction partners). The manual tasks are repre-
presented by these interactions that the user has with other people or with artefacts, such as books and manuals. These interactions interface with the model through the user and her activities in the electronic environment and the manual environment. The user forms in this way a natural interface to the manual environment. More specifically, the manual environment consists of colleagues (met face-to-face or via the telephone, for example), books,
paper, binders, calendars, ROLODEX, and other non-electronically available sources that affect both the user and the external sources in the electronic environment of the user. Also included here are equipment such as the telephone and the fax machine (if they are not accessed via the computer)—the manual environment includes sources that the user consults outside of her computer and e-mail system in order to carry out tasks.

Implicit in the model are manual tasks that interface with the electronic environment of the user. Manual tasks can modify external databases that are included in the electronic environment. That is, the manual tasks leave traces in the electronic environment—the user sees the results of manual tasks when she interacts with her electronic environment. If she has the right privileges she can also implicitly affect manual tasks by modifying the sources that the manual tasks are using.

As an example of the interaction pattern between the user and her manual and electronic environments, consider the checking of a meeting time. Typically, when time is of essence, the telephone is used to confirm that a message has reached its destination. In addition to the confirmation of delivery the user can check with the other party as to whether the meeting can be scheduled as suggested in the message. After the call the sender can mark the task as completed, possibly adding an annotation that a telephone call was needed. In this way information from the manual environment (the sender’s use of a telephone) is added to electronic environment. There is more to say about the electronic and manual environments, which we do in the next section.

A special module, the conversation support module, takes care of the user-controlled starting and ending of conversation logs, as well as the extraction of relevant conversation information that a user can request at any time. The log contains the history of an exchange of messages. The conversation information that can be extracted includes, among other things, users involved in the conversation, related documents, and deadlines.

The conversation information can be used by the user to check the conversation status, also called the task status. The status is represented by the messages, documents, and links communicated between the people and automatic systems involved in the task. It is from this that more information can be inferred, e.g., how many have answered a request for a meeting appointment, who have sent a certain report, etc.

The unit of conversation is used to preserve the task context, and to keep the relevant task information readily available and in one place. The task context is a combination of previously received messages, stored
message threads, people involved in the task and their roles in the task, databases employed, the personal calendar, etc.

A user may want to use templates for common tasks, in order to facilitate the composition of a message and to remind herself about the information needed to start a task. For example, in a request for a meeting, the user can be reminded to fill in a meeting time, duration, place, and agenda, among other things. A template can thus be used to represent a message type, e.g., a request for an appointment. Note that templates are not obligatory. However, if the user decides to use them, the information relevant for classifying messages will be more concrete and specific from the beginning.

A formalised conversation is an exchange of messages that has been predefined. This definition can include additional information, for example, other sources of information involved—and not just messages. This definition of additional information requires a structure. By giving the outgoing messages a structure, formalisation can be enabled. The structure contains more specific information about the conversation. The templates represent a natural way of introducing structure into the e-mail system. For example, the user can specifically state that the conversation concerns the task of scheduling a meeting. The conversation module is responsible for making templates available to the user.

The conversation support module also includes the handling of reminders, which a user initiates to remind her about deadlines. The reminders are handled by the ticker, which regularly checks the deadlines and alerts the user.

4.4.2 More about the electronic environment

The electronic environment is a collective name for different electronically available information sources of the user (This corresponds to the organisational memory discussed in Chapter 1). The information sources are heterogeneous and (often) distributed, i.e., they contain information in different formats, and they are managed separately from each other, possibly on different computers. We classify the sources in the electronic environment into the following classes:

• **personal sources**, which are mainly the desktop sources, the personal database (PDB) and the rules database
• **external sources**, for example a database of employees and project-related databases. Typically, the web-based intranet is included in this category (the organisational memory).
The sources can, e.g., be accessible through a separate piece of software, as an attachment to a previously received e-mail message or through a link in an e-mail message.

As the examples in the list above show, there can be many types of external sources, so we do not describe them here. However, the case is slightly different with regard to the personal sources, which we describe below.

**Desktop sources** The desktop sources include the address book, the calendar, and the web browser bookmarks of the user, as some examples.

**The personal database (PDB)** The personal database (PDB) mainly consists of the message archive. This DB includes the different folders and other groupings of messages found in the e-mail system. Included in this category are the outgoing messages, the current threads (conversations), and other non-threaded messages, typically arranged by subject or sender. Also included are the electronic annotations, if any, that the user has created for a message, either by adding extra text to the message body or by using an annotation software reminiscent of post-it notes.

**The rules database** The rules affect the classification and prioritisation of messages. The rules are created for the first time when a log of a conversation is started (see the middle part of Figure 4.3 on p. 59). The rules are subsequently modified and updated depending on what the user does in the electronic environment. When a task is finished and the user stops the logging of the conversation, the rules associated with the logging are made inactive in the rules database. The messages that make up the conversation are moved to a permanent store (a predefined folder, if not already in the folder, that is) for possible later references. The rules are available for later (re)use.

We now continue with two concrete examples involving the conceptual model: one example is based on the user’s view of the model while the other example is based on the task’s view.

### 4.5 A user’s view of the model

#### 4.5.1 Functional requirements

What kinds of functionality (services) are available to the user in the model? When we introduce the “new” concept of tasks, relating it to a conversation, into the domain of e-mail we must relate it to both the system services and the system functions that already are available. In this way the use of the concept, when it is actualised in the interface, will not be too difficult to understand by the user (Löwgren 1993).
The system services describe what the user can do with the system, and the user interface describes how the user can do it. The system functions realise the system services as perceived by the user (Löwgren 1993, p. 14). From a user’s viewpoint, the system services and functions can be described on both abstract and concrete levels. For example (cf. Takkinen 1994), in e-mail:

- the abstract system service named “communicate with others” is actualised by the abstract system function set containing “receive a message,” “send a message,” and “automate communication” (automation includes, for example, the automatic vacation message and/or rerouting of messages)
- the concrete system service called “compose a message” is actualised by the set of concrete system functions consisting of “fill in message header,” “write text in body,” “reply to a message,” “forward message,” and “append attachment.”

Note that there is no one-to-one correspondence between system services and system functions.

Apart from the basic services—consisting of functions for composing, sending, receiving, and reading e-mail—our conceptual model introduces a number of new functions required for full task management functionality.

To be able handle tasks a user needs to start, track, and end tasks. These are the three abstract system functions that implement the abstract system service of handling tasks (see Table B.1 on p. 296). On the concrete level, the services we need are start and stop log of conversation, along with request conversation information and program reminder. A listing of “old” and “new” abstract and concrete services in an e-mail system with task management are listed in Appendix B.

How do we then access the context of a task—and information related to it? As previously stated, we preserve task contexts by using conversations as the logical unit. The conversation acts as the key for access to the task context and contains information about the task: the users involved in the task, links to related documents, links to other messages and other information, etc. Switching between conversations enables the user to switch between contexts and having information about the current task readily available. Thus, handling conversations, and tracking which messages belong to which conversations, is important.

Reminders of deadlines should be supported, i.e., they should be programmable by the user so that the system can act on them. Since some task-related information can be coming from the manual environment, a means of entering this information into the electronic environment and the
e-mail system is needed. An example of this kind of information is contacts made by telephone that, for example, may indicate that a task has been finished.

To exemplify the use of the model from a user’s viewpoint, we use the previously presented scenario of scheduling a meeting (see Section 4.3 on p. 56).

### 4.5.2 The Scheduling-a-Meeting Scenario Revisited

We now return to the previously presented scheduling-a-meeting scenario, and add some details to it. A graphical overview of the scenario is shown in Figure 4.4. There are four people involved: the initiator of the task (person 0), person 1, person 2 and person 3. The dashed arrows going from the initiator to person 1, person 2, and person 3 represent the original request for a meeting sent by the initiator. The dashed arrows going from person 2 and person 3 to the initiator are automatic responses made by the e-mail systems when the users have opened the e-mail message (message disposition notifications, MDNs—see Figure 3.5 on p. 42). There is also another dashed arrow going from person 2 to the initiator, which is a personally written message commenting on the meeting request message. The continuous arrows between the initiator and person 1 represent a telephone conversation outside of the e-mail system. Finally, the dashed arrow going back to the initiator is a reminder, originally created by the initiator herself.

The full scenario is as follows. The initiator starts the scheduling-a-meeting task by first tentatively scheduling the meeting. She decides upon a meeting time and checks the availability of a meeting room by manually

![Figure 4.4: The scheduling-a-meeting scenario.](image-url)
examining the calendar available in association with the meeting room. Apart from making a note in the calendar associated with the meeting room, she also may (or may not) make a note about the tentative meeting in her electronic calendar of events that she has on the computer (cf. the electronic environment in Figure 4.3 on p. 59). She then consults her e-mail system and starts composing a message containing a request for a meeting. She decides to use a special message template for this. The initiator knows that by selecting a template it will cause the future conversation (the exchange of messages) on the message topic to be automatically logged (tracked) by the e-mail system. This means that outgoing and incoming messages concerning the meeting are going to be linked together and made available as an accessible unit in the Conversations folder in the e-mail system. The message template looks like an ordinary, empty e-mail message apart from placeholders for information about the meeting (see Figure 4.5). The participants of the meeting are put in the To: header and the main topic of the meeting in the Subject: header. After completing the template the initiator sends the request to the intended group of people. The rest of the text in the figure goes into the body of the message. She also selects to program a reminder to remind herself about the meeting (state e in Figure 4.6). The reminder can, e.g., either be an e-mail that will be sent by the calendar program or a separate message put on hold in the ticker folder in the e-mail system itself.

A state transition diagram in Figure 4.6 on p. 66 shows the detailed user’s view of the task. We use the representation introduced in Figure 2.3 on p. 19 and Figure 4.2 on p. 57, but with some additions. We disregard any message delivery errors. The circles represent the states and the filled arrows show the transitions between the states. The hollow arrows show documents that are either produced in a state or input to a state. The description of a state comes in two formats in the figure: an action to the left of the state and a log of the conversation to the right of the state.

| To: | Subject: |
| Date: | Time: |
| Duration: | Place: |
| Agenda: |

**Figure 4.5:** A template for a request for a meeting.
Figure 4.6: The user’s (initiator’s) view of scheduling a meeting in the model.
We only show the states where the task log is changed, e.g., composing and sending a message is one state instead of two as in the general Figure 4.2 on p. 57. States $a$ and $b$ represent the initial (empty) log, and the log containing the initiator’s original (single) request for a meeting, respectively. The initiator also programs a reminder (state $c$) in order to remind herself about the final scheduling of the meeting. In her next session at the computer, the initiator views the conversation and discovers that one person, namely person 1, has not yet read the meeting request. State $d$ in the figure shows this situation, where MDN stands for a message disposition notification message (see Chapter 3). During the time that the initiator is viewing the log, another message belonging to the same conversation comes in: person 2 has sent a response containing some additions to the agenda (state $e$ in Figure 4.6). However, person 1 has not yet responded, neither explicitly as person 2 (with a separately written message) nor implicitly as both person 2 and person 3 (via an MDN—see Figure 3.5 on p. 42). The initiator now waits. Two days before the meeting the initiator receives a reminder (see Figure 4.7) from the system (state $f$ in Figure 4.6). The initiator is hereby reminded about the conversation, reviews it, and sees that person 1 still has not responded. She decides to make a telephone call to person 1 to get an answer as quickly as possible (cf. Figure 4.6 on p. 66). Person 1 responds with a positive answer, and also comes with some suggestions to the meeting agenda. The initiator writes an annotation and adds it to the system, which adds it to the log of the conversation (state $g$ in Figure 4.6).

The meeting can now be scheduled permanently. The initiator makes a definite entry in her electronic calendar, after sending a confirmation message to the meeting participants (state $h$ in Figure 4.6). She then stops the

** Figure 4.7: An e-mail with a reminder from a calendar program. **
logging of the conversation by issuing the Stop Log of Conversation command (state $i$ in the figure).

Another example of the user’s view of the model, but based on a more complex task, is shown in Appendix A.

4.6 A task’s view of the model

4.6.1 Functional requirements from a system viewpoint

In this section we focus on the different states of the task, instead of the states of the user, as in previous section.

There is task-related information electronically available in the distributed and heterogeneous sources in the electronic environment. For example, information about the different people involved in a task can be found in the following sources:

- the outbox of the user’s e-mail system, containing messages with the e-mail addresses to the involved people in the To: headers
- the address book of the user, either available in the e-mail system or controlled by a separate program, containing the full names of people—and personal annotations about them, among other things
- the database of employees, centrally and externally available and containing information about people’s home and work addresses, affiliations, specialist knowledge, etc.

By cross-referencing these sources the information in an e-mail message can be related to a context—the information about a task can be enhanced. We have already stated that conversations are used to represent tasks. To be able to model tasks more formally a structure is needed that can carry task-related information, enhance the task context, and structure the conversation. By using the structure, task-related information can be made readily available. The structuring of the conversation makes it possible to forecast (to some degree) what is going to happen next in the conversation—who is going to act on the task.

As discussed earlier in this chapter, message templates can be used to introduce a structuring of the tasks. In order to make the construction of templates more general, a structuring language can be used, along with rules. It should be possible to reuse the templates, as well as the rules. The rules can be stored in the rules database and only made inactive when a task has been completed.
4.6.2 THE SCHEDULING-A-MEETING SCENARIO REVISITED

In this section we use a system view to look at how a task comes into existence, how it evolves, and subsequently is completed. We use the same scheduling-a-meeting scenario, and the instantiation of it, as in Section 4.5.

A state diagram representing the task-centred view of our model is shown in Figure 4.8 on p. 70. The circles represent the task states. The filled arrows with accompanying text represent the enabling activities, which are either activated by the initiator of the task or the system (the dashed arrow between states \( e \) and \( f \) is a special case, explained below). The hollow arrows denote input used in some of the states. The figure also includes the updated context of the task (shown to the left of the states in the figure). We only show the states where the context is changed.

The meeting scheduling task comes into existence when the Start Log of Conversation command is issued by the user (transition from state \( a \) to state \( b \) in Figure 4.8 on p. 70). The context is initialised to containing the task’s name and the users involved in the task (including who started the task). Also, the context contains a link to the current conversation, i.e., the exchange of messages that represents the task to the initiator. The task collects information about the newly started conversation—the scheduling of a meeting—from the personal and external sources available in the electronic environment (cf. the conceptual model in Figure 4.3 on p. 59). The calendar and the address book are both consulted and the information found there is cross-referenced with the information in the initial message (state \( c \) in the figure). The initial message contains the request for a meeting, which has been structured with the help of a message template. In our example, the initiator of the task also programs a reminder (state \( d \) in the figure). The task context is updated accordingly with a Documents section, containing a pointer to the reminder, and a Deadline.

Since the initiator of the task in our example has entered a tentative note about the meeting in her calendar, there is an entry in the calendar with the same date and time as in the initial e-mail message. More information about the receivers of the request for a meeting is found in the address book, such as their alternate e-mail addresses, affiliations, job titles, web pages, etc. The address book information can be used by the task to extract information from the external sources, such as the database of employees (if a receiver is situated in the same organisation, for example). A task context can now be created in which the task can operate. The task makes use of the information described above, along with information put in the initial
Figure 4.8: The task's view of scheduling a meeting in the model.
A Conceptual Model for Task Management

message partly by the user (via the template) and partly by the e-mail system (via the Internet e-mail protocol—see Chapter 3) to create the context.

The template for the initial message employs a standardised way to format and store the task-related information with a message. The structure and information are described for example by the use of a language. To be able to log the conversation, the task needs to keep track of incoming and outgoing messages that belong to the same conversation. Both the template and the e-mail system insert task-related information into the header and the body of the message which make the tracking of messages possible.

As can be seen in Figure 4.8, the task context also includes classification rules (state c). The task context is used by the task to create these classification rules. The rules are stored in the rules database. By collecting all related messages into conversations, the task context is made readily available for use and overview from one location in the system. The dashed arrow from state e to state d in Figure 4.8 represents the continuous classification of incoming messages during the task execution.

As an example of classification, the two MDNs in state d of the communication shown in Figure 4.6 on p. 66 can be inserted by the task into the right conversation thanks to e-mail protocol-specific information. This is in the form of unique message identities (IDs), contained in the References: and In-reply-to: headers (see Chapter 3). It is also possible to use this technique in most cases for the responses made by any of the receivers to a message sent by the initiator, which is the case in state e in Figure 4.6. The rule created in this case could for example check for matching IDs and then group these messages together.

However, there can be messages that belong to a conversation, but which do not contain any matching IDs. The task needs a way of relating these messages with other methods. A possible method is a function or a rule that checks the other information available in the header and body of a message and cross-references the information with that found in the electronic environment of the user. We discuss this problem more in detail in conjunction with the description of the prototype in Part III of the thesis.

The reminder has been handed to the ticker. The ticker (see Figure 4.3 on p. 56) checks the date and time of each reminder in its possession and activates them if the deadline has been reached. The ticker can, e.g., send an e-mail message to the initiator for the purpose to alert her about the task. When the message containing the reminder arrives (see state f in Figure 4.8), the task checks the headers of the message and sorts it accordingly into the right conversation.
The task is told by the initiator of the task to add a final annotation to the conversation (state $g$ in Figure 4.8). The annotation contains information about a telephone call the initiator made in order to be able to schedule the meeting in time. The task context is updated accordingly in the Documents section with a pointer to the annotation. In the next state (state $h$ in Figure 4.8) the task context is updated with the confirmation message that the initiator has sent to the meeting participants. The task then receives the Stop Log of Conversation command from the user (state $j$ in Figure 4.8). This indicates to the task to make the classification rules for the conversation inactive in the database. Also, the conversation itself containing the messages (and the annotation) is moved to a separate folder (state $k$ in the figure), for possible later reference or reuse.

The task-centred view of a more complex task, the making-conference-arrangements scenario, is available in Appendix A.

4.7 Conclusions

This ends the presentation and description of our conceptual model for task management in e-mail.

We note that the electronic and manual environments in our conceptual model constitute the organisational memory (OM) described in Chapter 1. The OM is the stored knowledge and experiences of an organisation. Researchers as well as practitioners recognise organisational memory (OM) as an important factor in the success of an organisation’s operations and responsiveness to changes and challenges in its environment (Stein 1995; Lehner & Maier 2000). The stored knowledge and experiences must be accessible in order to be useful. Ackerman and Mandel (1995) claim that OM information systems (OMIS) are only effective if they are designed to deal with specific tasks. In other words, information systems must not be seen isolated from the tasks they support. Rather, it is sensible to integrate them in every-day tasks. This is what Ackerman and Mandel (1995) call “memory in the small,” memory utilised in the effective performance of an organisational task. We follow this philosophy in including the OM when processing messages in the model.

The exchange of e-mail messages is the most central concept in the model and therefore we continue describing in more detail how the conversations are modelled. In the next chapter we describe a language that can be used both as a container for task-related information and as a means of structuring (formalising) a conversation.
Chapter 5
Formalising Conversations

In this chapter we focus on how to model and support the conversations, i.e., the exchange of messages. The conversations represent the tasks in our conceptual model described in the previous chapter. We start by first examining what we need—we introduce the concepts of message types and message templates. These are part of the mechanism for formalising conversations, which we specifically define and discuss. Subsequently we examine what types of automatic message handling systems are available for supporting conversations. The chapter is concluded with a description of our approach, which includes a language which uses speech act theory as a reference model. The use of the language is exemplified in the scheduling-a-meeting scenario.

5.1 What we need to support conversations
A central concept in the conceptual model presented in the previous chapter is the notion of conversations. As stated previously, we use conversations to represent tasks. A conversation in our model consists of the messages sent back and forth between two or more “message-processing entities” within a “task context.” The entities, or interaction partners, can be humans, computer programs, or both. The messages are normally related to each other, representing the turn-taking of the interaction partners in the communication.
In relation to the conceptual model, here we explore further the internals of the Conversation support module (see Figure 4.3 on p. 59). The Conversation support module handles the starting and ending of conversation logs, as well as the tracking of conversations. Also, we investigate the function of the Conversations folder in the model. The Conversations folder is used for storing current conversations, i.e., tasks, in one place, readily available to the user for immediate reference when needed.

We continue this section by examining what information and mechanisms we need in order to support conversations in the model. Thereafter we discuss the use of message types and message templates as a support for conversations. We conclude the section with explaining and describing our approach for supporting conversations.

5.1.1 AVAILABLE TASK-RELATED INFORMATION

Since conversations represent tasks in our conceptual model, we need to consider what information in a conversation is related to and relevant for a task. At least the following items should be included (cf. Chapter 2):

- the persons involved in a task
- the responsible person or software for a task
- the documents involved in a task
- the ordering of several tasks in time
- the starting and ending time for a task
- the subtasks of a task.

Some of this information is accessible through the standard Internet e-mail protocol (Resnick 2001) as part of the headers, e.g., the persons involved can be extracted from the From:, To: and Cc: headers. The e-mail protocol contains the following information about the conversation and the communicating parties involved in it (cf. Chapter 3):

- the sender
- the receiver
- the topic of the message
- the content of the message (text)
- the date and time for when the message was sent and when it was received
- the sending machine’s identity
- the identity of the previous message, if the current message is a reply
- the identity of the root message, i.e., the message that this exchange of messages started with.

74
Formalising Conversations

This information forms a core for the context of a conversation. A conversation has both a context and a status (cf. Chapter 2). The context consists of the relation between messages in a conversation, and the conversation’s relation to information sources in the electronic and manual environment (see also the previous chapter). The status of a conversation tells us who is in turn to send a message, and who have answered, among other things.

The messages need to be tracked so as to keep them within the right conversation and linked with relevant information. The tracking of conversations includes the handling of exceptions, i.e., occasions when the standard way of interaction is broken. Relating back to the conceptual model (see Figure 4.3 on p. 59), it is the Conversation support module that must be able to handle exceptions. A typical example of an exception is messages that do not contain any structural references to any previous conversation (message IDs, SAT information, Subject: header, etc.). Exceptions introduce uncertainty into the system and are the result of either the processing by the e-mail system or the interactions by the human user. We need to reduce this uncertainty. This can be done by introducing structure and other mechanisms so as to make the support for conversations more robust.

Message templates and message types form a foundation for the introduction of structure and support for conversations. We discuss these concepts in the next subsection.

5.1.2 Using Message Types and Message Templates

Messages can be classified as belonging to different message types. This classification can facilitate the structuring and automatic processing of messages, which is desirable in a task management system (cf. Chapter 1). One or more message types can be used to represent the main characteristics of a task, e.g., if the task is mainly about requesting information or disseminating information. Also, the message type can be used to initialise the task context.

There are a number of classification systems or schemes for e-mail messages based on message types. Maybe the most well-known use of message types to classify electronic messages is represented by Crowston, Malone, and Lin (1988). They analysed an organisation and defined a set consisting of four basic message types for describing messages exchanged in a computerised conference system—the message types are: Policy, Comment, Request for Comments (RFC), and Problem. In addition, two supplementary message types to represent messages that used the broadcast capabilities of the conferencing system were defined: Other/Classi-
CHAPTER 5

fied and Other/Unclassified (not otherwise classifiable). These message types were employed so as to examine organisational changes when the computerised conference system was introduced in the organisation.

Rice (1995) defined another classification system consisting of 13 message types, but only for a particular class of messages, namely *memoranda*. Memoranda are messages used for mainly disseminating information. Rice’s classes are: Announcement, Request, Explanation, Reply, Notice, Confirmation, Reminder, Instruction, Suggestion, Acknowledgement, Invitation, Offer, and Other. Rice used the classification for the purpose of examining how the e-mail medium influences stylistic choices in the language when composing messages.

Fleming and Kilgour (1994) interviewed e-mail users in order to find a classification system for messages—the following seven types were found: Problem solving, Negotiation, Arranging meetings, Group discussion, Information request, Information dissemination, and Informal. Their goal was to use the type of a message so as to determine how the information in a message should be displayed to a user.

Kimbrough and Moore (1997) identified seven message types for an Army office context: Read/Review/Comment, Appointment, Dissemination of information, Staff action, Query for information, Absence, and Statement. The list is only illustrative and not definitive—it was defined by repeatedly interviewing one knowledgeable user, a major in the U.S. Army, who also consulted other officers with relevant experience. The message types were developed and specialised for automatic processing of messages in the Army office environment, and they were shown to be useful to characterise the purposes of messages.

By implementing systems based on message handling procedures for each message type, the following hypothesis can be tested: that few message types exist, useful distinctions can be made, and procedures can be defined for handling each message type (Moore & Kimbrough 1994).

The message types can be used to define the task context. When a task is initialised the message types can play a central role in defining the type of support needed for the conversation. To make the support for conversation transparent to the user we need to employ *message templates*. The message

---

7. Messages of one of the following types: personnel (e.g., “Welcome to the group”), conference-system-related, announcements of job openings, personal (e.g., “Merry Christmas”), thanks for some answer, meeting or meeting attendance announcements and questions (e.g. “Can we not meet on Sunday?”; “No, we have to meet then.”).
templates are based on the message types and define the initial task context. The user can also be explicitly reminded by the template about what information is needed to define and start the task, e.g., that a meeting time and place (among other things) must be provided in a request for a meeting.

Moreover, we need a language for describing what we can do and not do with the messages, the conversations, and the information related to the messages. Each message type represents a specific type of conversation and use of the language. The message templates introduce the language into the system, at the same time hiding the language from the user. In other words, when the user completes a template she at the same time implicitly selects a message type and thereby uses the language to initialise the task context and the formalisation of the conversation.

Finally, we need a mechanism for processing the messages and for managing the conversation. The mechanism includes the automation of some parts of the communication and/or task management. The automation can, for example, be in the form of rules that trigger on message types so as to filter, sort, and prioritise incoming messages into folders. The message types described above are only one part of the automation.

To summarise, we show the relationships between the concepts introduced in this section in Figure 5.1. These concepts are all part of the Con-

![Figure 5.1: The supporting relationships between the concepts introduced in Section 5.1.](image-url)
Chapter 5

Message templates support the creation of messages and message types. In other words, message types are introduced into the system by the message templates. Moreover, the message types support the automation of messages and conversations. The automation, supported partly by the message types, supports messages and conversations. This means that the automation helps in tracking the messages belonging to the conversations.

In the next section we examine the available types of systems for automating the handling of messages.

5.2 Automating handling of messages

There are four general approaches to automated message handling (cf. Kimbrough & Moore 1997):

1. natural language processing
2. standardised-document system
3. tagged-message system
4. declarations-in-message system.

The message handling functionality includes the definition and organisation of messages, and possible inferencing of information. We discuss each of the approaches below and how each of them handles conversations.

5.2.1 Natural language processing

The natural language processing (NLP) approach includes techniques that go beyond the keyword or statistically (word-counting) based approaches characteristic of traditional information retrieval (IR). IR research has traditionally focused on the automatic indexing and retrieval of documents based on keywords (see, e.g., Salton 1989). The keywords have been treated as occurring independently of each other in a document—disregarding the context of a word.

NLP, compared to traditional IR, incorporates the context of a word in the message handling process, and not just the word exclusively. The context is either a phrase, a sentence, a paragraph, or a section. However, systems that can generate in-depth analyses of unconstrained text—and do it with computational efficiency—are not yet practicable. Instead different algorithms are employed that examine documents for the presence of varying numbers of linguistic expressions, surrounding contextual information, and combina-
FORMALISING CONVERSATIONS

tions of key phrases. This simplifies the job of an NLP system because portions of the text not relevant to the domain can be effectively ignored. Furthermore, in limited domains and with domain-specific dictionaries, NLP can be usable in practice (see, e.g., Young & Hayes 1985).

A representative of the NLP approach is the method of information extraction (IE). IE assumes that a full grammatical understanding of the text is not feasible or even necessary. That is, natural language sentences need to be understood only partially when the intention is to extract (discover and store) specific kinds of information from a source document.

The user of an IE system normally knows what she is searching for, and her goal in using an IE system is to obtain a more “intelligent” answer. The output normally consists of a summary of the contents of the document in the form of a template (Jacobs 1990). This template in IE is a structure with a predefined number of slots, which are filled with the information extracted from the source document. For example, in an IE system scanning financial news stories to identify and extract details from those dealing with takeovers, a template named “takeover” would have the slots “Company target,” “Company predator,” “Type of takeover,” and “Value” (Costantino, Morgan & Collingham 1996). The information in the slots is either directly extracted from the document or inferred by the IE system from data that represent the meaning of the document. In the latter case, the slot “Type of takeover” in the example above could thus be filled with the term “Friendly,” although the term itself was not found in the document. In other words, the IE system can retrieve documents that are both directly and indirectly related to an initial document.

An example of an advanced IE system is the CIRCUS sentence analyser described by Riloff and Lehnert (1994). Though it is domain-specific, the specific IE method in CIRCUS (actually consisting of three different algorithms) can be ported to other domains. This can be done by acquiring new domain-specific dictionaries (a labour-expensive task) and a domain-specific training corpus—this part can be automated to a large extent (see Riloff & Lehnert 1994 for details). McCallum and colleagues (1999) use Hidden Markov Models (HMMs), widely used for speech recognition and part-of-speech-tagging (Charniak 1993), to identify headers and reference sections of computer science research papers that are to be classified into a topic hierarchy used by a search engine on the web. Boisson and Shahmehri (2000) have investigated how the information in a text document in an electronic publishing application can be interpreted as instructions to pursue a given task. The information pattern thus found can be seen as the
triggering mechanism for a set of predefined operations, often repetitive tasks. Another (simple) example of IE is the detection of proper noun phrases in CLUES (Marx & Schmandt 1996), a message filtering system for e-mail: The proper noun phrases are determined by consecutive capitalisation.

The COSMA (Cooperative Schedule Management Agent) project at DFKI, Saarbrücken (see “Project COSMA” 1997), uses NLP to automate the scheduling of appointments via e-mail. Cooperating software agents negotiate and add appointments to calendars according to their user’s instructions. Participants without agents can communicate in natural German language—COSMA analyses these messages using state of the art language technology and so-called shallow parsing techniques. A message extraction system called SMES is used to produce predicate argument structures containing shallow semantic analyses of predicate phrases and noun phrases (Busemann et al. 1997).

In natural language, conversations are traditionally thought of as having a beginning and ending. Through the discussion in a conversation a conversational context evolves (Whittaker, Frohlich, & Daly-Jones 1994). With regard to informal communications, openings and closings are often missing. Furthermore, informal communications seem to consist of one long intermittent conversation consisting of multiple unplanned fragments. Normally, informal communications are defined as taking place synchronously in a face-to-face setting. Furthermore, natural language is characterised by many exceptions from “the standard case,” i.e., many communication errors occur, which have to be handled in an effective way if natural language is to be processed in a system. These are all factors that a computer has to be programmed for if it is going to be able to process natural language and make use of it.

In conclusion, computers that read and analyse—use full-scale NLP—are not generally viewed as usable in practice, although in limited domains and with domain-specific dictionaries they can be useful, as exemplified above.

5.2.2 Standardised-document system
This approach is represented by Electronic Data Interchange (EDI). EDI focuses on automating routine business transactions. It is the direct transfer of structured business data, i.e., common business documents, between computers in a standard format and by electronic means (cf. “The XML/EDI Group’s Home Page” 2001). EDI consists of several protocols that have been developed for computer-based processing of, for example, solicitations for bids, orders, invoices, and payments.
The term **transaction sets** is used in EDI to refer to the electronic version of the business or technical information that trading partners exchange via EDI. It is a standardised formatted message that is the electronic equivalent of a business paper document. The user encloses transaction sets in electronic envelopes, and ships the documents to the intended receiver. Some examples of transaction sets are (see the ANSI X.12 standard in Diffuse 2001):

- purchase order
- invoice
- ship notice
- Request for Quotation (RFQ)
- quote.

The standard mostly used in North America for implementing EDI is ANSI X.12. It is similar in nature to EDIFACT (EDI for Commerce, Administration and Transport), which is the *ISO standard* for implementing EDI for commerce, administration, and transport.

An example of a valid message for the transaction set RFQ in X.12 is shown in Figure 5.2, along with an explanation in English of this message (Kimbrough & Moore 1997). For an organisation, becoming EDI capable requires change in many areas. EDI redefines and changes the ways in which a business operates and relates to other businesses. This implies that EDI is good for what it was designed for—the document-orientation means that EDI is designed to directly replace specific types of standard paper documents with each variant of the protocol. This also means that EDI is inflexible and document-oriented—messages must be standardised business documents—as opposed to message-oriented, where messages can be unstructured. These are aspects that make the standardised-document approach unsuitable for our purposes.

### 5.2.3 Tagged-message system

A typical example of a tagged-message system is the traditional Internet e-mail system with its e-mail headers and body, and the information contained in these (see Chapter 8). The headers function as **tags** of the messages. New
headers can be added by the user, so-called X-headers—two examples are X-url: and X-mailer:. The former contains the url for the homepage of the sender, while the latter contains information about the sender’s e-mail reader. Compared to standardised-document systems (EDI systems; see the previous section), tagged-message systems pass messages of low formality. Processing systems are created that enable users to process the tags in order to filter, sort and prioritise messages. The content of the body of messages is not normally available for processing, so the only information available is contained in the headers.

An early representative of this approach is INFORMATION LENS, an “intelligent tool for managing electronic messages” (Malone, Lai, & Grant 1997). It helps people filter, sort, and prioritise messages. INFORMATION LENS uses semistructured message types (based on frames or templates) and sets of IF–THEN rules for automating the processing of messages (Malone et al. 1987a). A hierarchy of semi-formal templates is used to structure messages. The templates are designed to aid the user in structuring a message depending on the purpose of the message. Furthermore, the structure is used in the selection, sorting, and prioritisation of received messages. Also, the structure can be used for generating automatic

Figure 5.2: An EDI Request for Quotation (RFQ) (above) and its English translation (below) (from Kimbrough & Moore 1997).
answers and to suggest different answers to a receiver. A template for making a meeting appointment in INFORMATION LENS contains fields for time, place, speaker, and subject. It is also possible to define one’s own templates. The IF–THEN rules provide a means to automatically move messages into folders, remove messages, and to forward messages to other users.

The semantics of the contents of messages in a tagged-message system is not normally available for processing. Furthermore, the tags are not defined in a recursive fashion (Kimbrough & Moore 1997); for each new tag a new symbol must be defined. For example, a message inside another message would have to be expressed as a new tag in a tagged-message system. Here, a language would provide a general mechanism for expressing this.

5.2.4 DECLARATIONS-IN-MESSAGE SYSTEM

In this approach the header of a message is expressed as a data structure and contains declarations about the message. The declarations can be used for inferencing about the contents and the context of the message, and the conversation it belongs to.

The formal language for business communication (FLBC) described by Kimbrough and Moore (1997) is an example of this approach. The language is flexible in that it can be used to express more than what is possible in the tagged-message system (see previous section) in a computer-based conversation, e.g., storing references to a message within another message. The language can also be extended. It is structured using speech act theory (SAT), which also provides for the flexibility. SAT is a general and fundamental theory for linguistic communication. Its use is illustrated by the following citation, attributed to Wittgenstein (Allwood & Andersson 1984): “Don’t look for the meaning of a word, look for its use.” We discuss SAT more thoroughly in Section 5.3.3 on p. 90. However, basing a language on SAT and using it as a basis for designing information systems is not unique.

Winograd and Flores (1986) describe THE COORDINATOR, which actually uses a combination of tagged messages and SAT. It represents an early e-mail-based system that helped people structure conversations and track tasks based on SAT. Here, a typical “Conversation for Action” begins with a “Request” message from the sender, explicitly requesting the receiver to do something by a certain date. The receiver is then prompted to respond with either of the following (Malone, Lai, & Grant 1997):

• a “Promise” message, promising to perform the action
• a “Decline” message, declining to perform the action
• a “Counteroffer” message, offering to perform the action by a different date or to perform a different action.

If the receiver promises to perform the action she may then, for example, later send a “Report completion” message, stating that she indeed has performed the action. The original sender then may find this action satisfactory and finish the task by sending a “Close” message. Apart from these specific forms of communication, THE COORDINATOR also provides a default “freeform” message construct.

THE COORDINATOR is recognised as one of the first groupware systems ever built. It was evaluated at Pacific Bell in a study that revealed that the system was “unnatural” and “uncomfortable” (Robinson 1991). It also required extensive effort to learn how to use it (Grudin 1988, p. 87). THE COORDINATOR failed mainly because it did not support “open-ended, free-flowing, almost serendipitous conversations” (Robinson 1991). However, THE COORDINATOR succeeded in contexts which were provided by “stable, hierarchical, authoritarian organisations” (ibid.), such as the Army office context. In conclusion, THE COORDINATOR was not suited for e-mail (spontaneous communication), but well suited for military office communication, where user ranks are clear.

By using SAT it is possible to express more and also do it with more flexibility, compared to traditional tagged-message systems, but still keep it within a formalised context from a computing viewpoint (Kimbrough & Moore 1997). The language enables a limited interpretation of messages by machines and the automated extraction of information. Furthermore, it is possible to iterate message operators, i.e., a message can contain another message. For example, we can express that “Smith said that Jones requests that Smith be put on the subscription list of this mailing list” (cf. Kimbrough & Moore 1997). Compared to the tagged-message system approach above, the declarations-in-message system approach already contains all the basic elements for all types of communication. In other words, there is no need to define new sets of elements, such as the X-headers used in e-mail (see Chapter 3).

5.3 Introducing a language for tasks in e-mail

We have elected to use the declarations-in-message system approach described above for modelling the conversations—and consequently the tasks—in e-mail. We see this approach as a natural extension to the tagged-message system approach, i.e., the Internet e-mail systems in use today. The
iterative construct in a language such as the FLBC described in Section 5.2.4 above can be used to define conversations that model composite or delegated tasks, such as making conference arrangements (see Appendix A). This is possible but not feasible in the tagged-message system approach.

With regard to the message types (see Section 5.1.2 on p. 75) the messaging environment influences what kind of message types and how many are needed. The messaging environment also influences how the message types are processed and used. As Moore and Kimbrough (1994) have pointed out, it is impossible to foresee all the message types that might be needed. This implies the need for a flexible language. We use the eight message types listed in Table 5.1 on p. 86 as a starting point.

The first seven message types listed in the table were originally created by Kimbrough and Moore (1997) for an Army Office environment. We have extended the set of message types with an eighth category Other (OTH), so as to cover messages that are not possible to classify as any of the other types.

Before we describe the details of our language we first take a look at an example of a simple language. We also discuss why and how to formalise a conversation, and introduce the concepts specific to speech act theory, which is a theory for communication that we use as a reference model for our language.

5.3.1 AN EXAMPLE OF A SIMPLE LANGUAGE

We show here an example of a formalised conversation and the use of a (simple) language in the communication between a human user and a mailing list manager (MLM) program.

A mailing list is a way of having a group discussion by e-mail and distributing announcements to a large number of people. All of this traffic is automated and managed by the MLM. The MLM copies incoming e-mail messages to all members of a subscription list maintained by the MLM. To become a member of the list a user must send a subscribe message to the MLM, along with the name of the list. The MLM puts the user's e-mail address on the subscription list and answers with an acknowledgement. Each time any list member posts a message or a reply to a message on the list, the message is distributed to the e-mail box of every member of the list.

The two most frequently used programs are LISTSERV ("L-Soft international" 2001) and MAJORDOMO (Chapman 1992). For example, to

---

9. An evaluation of the message types—and if they fulfil the needs we and the users have—is beyond the scope of this thesis.
subscribe to Dance, a mailing list on a MAJORDOMO MLM at the author’s department, the following steps are taken (see also Figure 5.3):

Table 5.1: Eight message types for office communication

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS: Absence</td>
<td>This type is used to give notification of planned and authorised future absences. This enables future messages to be rerouted to alternate personnel who are not absent.</td>
</tr>
<tr>
<td>APP: Appointment</td>
<td>Requires the receiver to respond to the sender’s request. The response can be elaborate, especially if it is negative and an explanation is required along with the “no” answer.</td>
</tr>
<tr>
<td>DOI: Dissemination of information</td>
<td>This is a “send and forget” message, seen from the sender’s point of view. Here, it is the responsibility of the receiver to read and (not) act on the message.</td>
</tr>
<tr>
<td>QFI: Query for information</td>
<td>This type is closely related to the SAC message type above. The information requested is relatively easy to retrieve by the receiver, while the processing of the information is included in the SAC message.</td>
</tr>
<tr>
<td>RRC: Read/Review/Comment</td>
<td>Provides the sender with the capability to distribute documents and messages to other users, and to assign one or more people to read, to act on (if appropriate), and to critique a document. Optionally the sender can require a response and set a date and time when some specified action is to be completed. Acknowledgements may also be required.</td>
</tr>
<tr>
<td>SAC: Staff action</td>
<td>Here, one or more responses are required by the receiver. Typically, the required response includes the requested action, for example, when a report is to be written and delivered by a particular time. Moreover, the sender of an SAC message usually desires more responses, such as message receipt confirmation and acknowledgement of intermediate due dates (milestones). These responses must be specified explicitly by the sender.</td>
</tr>
<tr>
<td>STM: Statement</td>
<td>This type of message is used to convey information in much the same way as a DOI message. However, an STM message is an assertion by the sender, to the receiver, that the content of the message is in fact true, which discriminates it from the DOI message.</td>
</tr>
<tr>
<td>OTH: Other</td>
<td>This type is used for information that does not fit any of the other categories—which cannot be classified.</td>
</tr>
</tbody>
</table>
1. The user sends an e-mail message to \texttt{majordomo@ida.liu.se} with the following text in the body of the message:

   \begin{verbatim}
   subscribe dance
   end
   \end{verbatim}

2. The MLM receives the subscription request, puts the e-mail address on the subscription list, and sends a confirmation (possibly with information about the list) to the user.

The MLM has a limited set of commands that it “understands,” typically the two commands \texttt{subscribe} and \texttt{unsubscribe}, in addition to commands relating to the administration of the mailing list. A person is usually responsible for the list. For the MLM to be able to parse the commands, only one command can appear on each line in the body of the message. The MLM can be said to use a simple language, consisting of commands and the responses that should be given to these.

This MLM language can be used for one specific task—the task of subscribing to a mailing list. This language is also very rigid. There is much more that we want to say in a conversation, and not just in an ad hoc way. For example, when scheduling a meeting via e-mail we also would like to be able to state the following in a more formal way:

- who should respond to the request for a meeting and to whom
- when is the deadline for responding, and when and how are reminders handled
- who have responded and who have not, \textsl{etc.}

The third point above concerns the status of the conversation (\textit{cf.} Section 5.1.1 on p. 74). It also exemplifies a type of information that may not be readily available and thus must be inferred from other information.
To be useful for other tasks the language used in the MLM program exemplified above would have to be extended. In a language, we need to be able to specify different types of tasks. The message types described earlier (see Table 5.1 on p. 86) represent the different tasks we would like to model in the language. With this requirement comes the need to store the context of the different tasks. In other words, we need a language which for each message type can describe the context for the task related to the message type. Furthermore, since e-mail-based task management in general is very ad hoc in its nature, the language must provide an “escape hatch” from the rigidness that comes with the use of a language. Thus the language must be able to handle cases where a message type is not defined.

To summarise, the language must be simple, yet not too simple, in addition to being flexible. With flexible we mean a language that is expressive and extensible. By using a flexible language we can say more in a conversation (cf. Kimbrough & Moore 1997). When we can say more we can make it possible to infer more information about the conversation. 

Speech act theory (SAT) forms a good foundation for reasoning about conversations. Though controversial, SAT is widely accepted in linguistics, philosophy of language, and artificial language. Before we describe how we make use of SAT, however, we need to examine what formalisation means in the context of conversations in general, and specifically, what it means in the context of a language for conversations.

5.3.2 Formalisation through a language

There is a procedure employed by the users to handle the messages in an e-mail system. This procedure determines to a large degree the rigidness that should be imposed on the exchange of messages (the conversation). The relation between the users determines the procedure. In an Army Office, for example, there is a strict hierarchy between users, which is also mirrored in the procedure that handles the message types (Kimbrough & Moore 1997). Formalisation of the conversation facilitates the automated processing of messages. By using a flexible language the processing can also become more flexible.

The language provided by the Conversation support module must reduce the uncertainty so as to make automatic message handling possible. By using a language, the information from an Internet e-mail message (cf. Section 5.1.1 on p. 74) can be collected and stored in the structure provided by the language. Other types of information that we need to model in our language relate to the view of a conversation as being a task. For example,
the language can be used for making information relating to the current task readily available, either explicitly by including the information in the asynchronously sent messages, or implicitly by providing links to the task-related information. In this way the task context can be taken into account when processing messages. The language can be utilised to check the status of a task and, for example, see whose turn it is to act in a conversation.

We note that many systems that require extended formalisation of procedure and interaction, or of structure and content, have many problems (Shipman & Marshall 1999). Users are not very willing to use the formal aspects of the systems—cf. THE COORDINATOR (Winograd & Flores 1986) and INFORMATION LENS (Malone et al. 1987a). Especially systems that are coordination-oriented, such as groupware, also often try to formalise social practices which normally are implicit in human–human interactions. This kind of formalisation is often difficult since it is impossible to make the implicit rules explicit—it is also undesirable from a social point of view (cf. Shipman & Marshall 1999). There is a portion of informal conversation and social practice that makes the formalisation of e-mail communication less successful. Here a system that allows formalised and non-formalised tasks to coexist would be the preferred. Also, users do not want to describe their procedures for whether and when to schedule a meeting with other people (Grudin 1988). The same rules that one uses for scheduling a meeting with one’s boss do not apply to a total stranger. Differentiating rules is difficult and also socially undesirable.

However, computers require formally represented information to support users but users often cannot provide it. A simple example of a required formalisation is the order of characters which a word processor must know (cf. Shipman & Marshall 1999). For a look at the nature of this problem and a computer-based support for incremental formalisation, see Shipman and McCall (1994).

In conclusion, by specifying and using a language we get a tool for organising and defining tasks, in addition to managing tasks. Specifically, the language facilitates the automated handling of tasks. We also get a platform for inferencing information. The language functions as a container for both task-related information (including links to this information), as well as an implicit and indirect support to the user for formalising her tasks. The implicit and indirect support is necessary, since we want to keep the simplicity and spontaneity of e-mail use. Although the language supports the user in her formalisation of her tasks—it helps the user in structuring the conversation and is also accessible to her—the lan-
guage should in practice be mostly hidden and mainly used by the system in the background. The reason for this is, again, simplicity.

We stress the fact that it is important that we do not lose the informal communications and the spontaneity inherent in e-mail. Thus, the language we specify must include the handling of these informal communications, for example by being in itself less rigid or making it possible for the user to “go outside of the system.”

5.3.3 INTRODUCING SPEECH ACT THEORY

If we examine an e-mail message composed by a human user more closely we can recognise that it has the following items:

- a sender
- a receiver
- a content, hopefully related to the Subject: content
- a context—relation to previously received messages
- active/passive verb forms and other stylistic choices in the text (cf. Rice 1995); smileys, large caps, and other de facto standard symbols; which all give clues to the attitude of the sender.

As it happens, these are also components that need to be identified when using a basic strategy for representing a speech act in speech act theory (SAT) (Austin 1962; Searle 1969). However, SAT uses another set of terms: the sender is called the speaker, the receiver is the hearer and the verb forms, smileys, etc. represent (partly) the attitude or the illocutionary force.

In the domain of e-mail we have automatic access to all of the items listed above, except the last component—the illocutionary force must be provided by the user and is typically difficult to determine in an automatic way. The illocutionary force represents what the sender is doing in sending the message, such as asserting, requesting, or denying something. It can be introduced into the system via the use of message types, which in turn are defined through the use of templates. If a template is not used, the illocutionary force will be unknown.

SAT assumes that the minimal form of communication between people is not a sentence or other expression, but the performance of certain kinds of language acts. In other words, an act of linguistic communication is an act of expressing an attitude by means of saying something (Bach & Harnish 1979), i.e., to say something is to perform an act. Examples of these acts are requests and promises.
FORMALISING CONVERSATIONS

The speech act scenario (cf. Bach & Harnish 1979) begins with the utterance act. The utterance, $u$, can be a sentence, a fragment of a sentence (for example, “She is in the [pointing to the meeting room]”), or a sign designating a sentence (for example, “thumbs up” to accept an offer to sell stock). The context, $c$, can consist of certain conventions and assumptions (for example, that English is the primary language), certain gestures and inflections of speech (such as pointing and using emphasis when speaking), relevant history associated with the conversation (for example, locate the reference to a pronoun), and relevant ambient facts (for example, “We’ll meet in half an hour,” which means the speaker will meet the hearers at 5.00 P.M. if the time of the utterance act is 4.30 P.M. What to put into a relevant context in a given situation is not always completely clear, however, at least the history of a conversation should be included (cf. Kimbrough & Moore 1997).

The locutionary act is the second step in the speech act scenario. The hearer, $h$, has heard the speaker, $s$, utter $u$ in the context, $c$. The hearer must now figure out (infer) what the utterance means. The locutionary act succeeds if the hearer understands the content of the speaker’s utterance the way the speaker intended her utterance to be understood.

The third step is the illocutionary act. Here, the hearer has heard (or observed) the utterance act and has successfully interpreted it. The illocutionary act represents what the speaker is really saying, i.e., the attitude that the speaker has: Is she making a prediction, a promise, a lament, etc.? If the hearer correctly infers the attitude, the act succeeds. If the hearer does not infer the attitude correctly, the illocutionary act does not succeed, and therefore she will be still in the second step.

The fourth and final step in the speech act scenario is the perlocutionary act, which includes the effects that the speaker’s utterance has on the hearer. For example, if the illocutionary act is a promise then the hearer may rely on it and cancel another commitment in order to fulfil the speaker’s request.

Many illocutionary acts produce effects rather than develop understanding (Austin 1962), such as, for example, the acts of commanding someone to get out, warning someone to look out, or appointing someone to a committee. These are examples of acts that represent action. Our focus is therefore on selecting a language for the utterance acts that can be used to express the illocutionary acts that are needed for task management in e-mail. Kimbrough and Moore (1997) have defined a formal language for utterance acts that can express the illocutionary acts needed in conduct of
commerce. They used SAT in order to provide the system with access to the semantics of messages.

SAT was originally developed from the analysis of communication where the spoken word is used. However, SAT is interesting to use when analysing communication that happens through other media, such as information systems and paper documents. SAT has also lead to the research area of Language Action Perspective (LAP—see Flores & Ludlow 1980). The combination of SAT with tagged e-mail messages promises to provide the basis for expressively rich computerised messaging systems (Kimbrough & Moore 1997). However, we hinted at problems with SAT in an office context earlier (see Section 5.2.4 on p. 83)—we return to the discussion of these problems, and also LAP, when we relate our work to other work in Chapter 6.

We use SAT as a reference model for our language, a Formal Language for Conversations (FLC). SAT provides us with a model for how a flexible communication can be structured. We are not striving to make all of the communication formal, even though the language itself is formally defined. To simplify, the formal part of the language is needed for information retrieval purposes.

The idea is that the language (FLC) can express what the message does, what it is about, and essential elements of its context. The message itself is in the language and is therefore not tagged by the FLC description. FLC—similarly to FLBC (Kimbrough & Moore 1997)—gives us expressive power, as opposed to what we can express in the context of the MLM program described in Section 5.3.1 on p. 85.

Next, we describe the language FLC by defining the vocabulary and grammar that constitute it.

5.4 A Formal Language for Conversations (FLC)

In order to describe our language—a Formal Language for Conversations (FLC)—we first need to define the set of terms that can be used in a message, i.e., the vocabulary. The vocabulary consists of objects, illocutionary forces (attitudes), and predicates. We also need the means of putting the terms together to form a valid message, i.e., the grammar.

10. We have named our language FLC, which is reminiscent of FLBC (Kimbrough & Moore 1997), so as to show the relationship with FLBC.
When we have defined the vocabulary and the grammar, we conclude this section by showing how the different message types presented earlier (see Table 5.1 on p. 86) can be represented in the language. (The subsequent Section 5.5 concludes the chapter with a detailed example of FLC within the context of the scheduling-a-meeting scenario.)

5.4.1 The vocabulary of the language

From the discussion in Section 5.1 and Section 5.2 we find that we need components in the vocabulary from the following three areas, representing three points of view:

- message-centred
- task-centred
- speech act centred.

The vocabulary of the language, and its objects, attitudes, and predicates, can be seen in Figure 5.4 on p. 94. The attitudes and the predicates are listed at the bottom of the figure. According to the different views listed above we get different kinds of components in the language.

The message-centred view gives us the simpler objects in the vocabulary, such as `<person-id>`, `<message-id>`, `<document-id>`, and `<string>`. The latter is used for any text, including the body of a message. As another example, the Date: header of a message, containing a value of for example 9.33.44 for the time and September 17, 2001, for the date, is represented by the component `time(2001, 9, 17, 9, 33, 44)` (in our language the `time` construct also includes the date) and `date(2001, 9, 17)` (line 1 in Figure 5.4).

Looking from a task-centred view we need to be able to specify the order of activities. This is done by using `<time-pred>`, which can be employed to define activities that occur either before, after, at, on, or between certain dates and times (see line 4 in Figure 5.4). In addition, a task can be given a priority between 1 and 5, with 1 being the highest level of priority. This is done by using the `<priority>` component in conjunction with a predicate (described in the speech act centred view below).

There is also a specific predicate named urgent, which corresponds to priority level 1.11 Also belonging to the task-centred view, the predicate

11. The meaning of the urgent predicate can be inverted with the `isNot` component in the `<content>` (see the grammar). The use of priority is not obligatory and either of the two systems (priority levels 1–5 or urgent/not urgent) can be employed.
Chapter 5

Figure 5.4: The vocabulary of FLC.

**actionRequired** (see line 13 in the figure) is used for marking messages as requiring some kind of action within a task context. This component can be used to create to-do lists, including a **priority** (line 10) that can be set for a message, implicitly representing the priority for a whole task. The **actionRequired** component typically also contains a **deadline**, which can be defined using **time-pred** mentioned above, and a **note** (defined using **string**). Reminders are also central for task management—they are represented by the predicate **reminder** (see line 13).
One of the parameters in the reminder component is the time for when the reminder should be sent. We describe some of the other predicates in the speech act centred view next.\footnote{The details of each of the individual predicates and their arguments are implementation-specific and therefore not described here. See Chapter 8.}

From the \textit{speech act centred} point of view three different illocutionary forces (attitudes) are modelled in the language, using the component \texttt{<attitude>}, namely \texttt{assert}, \texttt{query}, and \texttt{request} (see line 12 in Figure 5.4). These are the only attitudes used in our language for now, in addition to the attitude \texttt{unknown}, which is used when no attitude is given by the user. Other attitudes can easily be added, should we find that we need more. The \texttt{<predicate>} component in the vocabulary (see line 13), finally, complements the \texttt{<attitude>} component. It represents the content of the message. The \texttt{actionRequired} component, which we listed above as belonging to the task-centred view, actually also partly belongs to the speech act centred view: The sender \texttt{assert}s (which is the \texttt{<attitude>}) that an action is required (which is represented by the predicate \texttt{actionRequired}) with regard to the contents of the message. Considering another example, when the sender is merely disseminating information she assumes the information being sent is \texttt{interesting} to the receiver(s). In terms of the language the sender \texttt{assert}s (the \texttt{<attitude>}) that the information is \texttt{interesting} (the \texttt{<predicate>}) to the receiver(s). Note that the same predicate can be used in conjunction with two different attitudes. For example, a sender can be either \texttt{disseminating} or \texttt{wanting a copy} of “the HICSS-32 conference paper by Takkinen and Shahmehri.” Here we have two different attitudes, \texttt{assert} (“disseminating something interesting”) and \texttt{request} (“wanting a copy of something”), but with the same predicate (“HICSS-32 conference paper”).

The \texttt{forwardedFrom} predicate is used when forwarding messages, which is a common activity in e-mail. Two parameters are central for \texttt{forwardedFrom}: the message ID of the message being forwarded and the person who is doing the forwarding.
5.4.2 The Grammar of the Language

Regarding the vocabulary presented above, we need to specify how the different kinds of components in the vocabulary can be used. The grammar—or the “language”—breaks a message up into three major parts, namely the force (the attitude described above), the content (the body of the message), and the context.

Referring to the grammar is listed in Figure 5.5 on p. 97, a <conversation> consists of at least one <msg-st>. The <msg-st> component consists of <sender>, <receiver>, <attitude>, <content>, the <context>, and <message id> (line 2 in the figure). At first glance, it may seem that only one receiver of a message is allowed in the language. As a matter of fact, in SAT one message “token” can only be sent to one receiver at one time (Kimbrough & Moore 1997). For e-mail purposes we need a way of expressing that a message can be sent to more than one receiver. This is accomplished by the use of the component alsoSentTo in the <context> (Figure 5.5, line 6). In other words, if the To: header of the e-mail message contains more than one receiver, the first one becomes the <receiver> in <msg-st> and the rest are listed by the alsoSentTo component. The message sent to the rest of the receivers listed in alsoSentTo have the same date and time as the current message token and is identical in attitude and content. The alsoAddressedTo described below alsoSentTo in the figure, by the way, is used in a special case of multiple receivers, namely for “carbon copies,” i.e., receivers listed in the Cc: header.

The <pred-st> in the <content> component introduces the predicates described in the vocabulary above (see line 3 in Figure 5.5). To provide for the possibility to define “tasks in tasks,” i.e., a task consisting of subtasks or nested tasks, we also need a mechanism for specifying nested messages. A special case of the use of this component is the forwarding of another message in e-mail (including a message in another message). This we can accomplish with the <msg-st> component in the <content>. It is also possible to include only the basic message structure in the content of a message—the simpleUtterance component is used for messages in which context and message identifiers need not to be defined. This is useful for embedded messages, such as “A says that B says X,” where “B says X” is treated as a simpleUtterance. Another example is “A says

---

13. This is to accomplish for very simple conversations, such as, “disseminating information,” which is a one-way (and often) read-and-forget type of communication.
that B must report to A about X” (typically after a specified time period),
where “B must report to A about X” is the simpleUtterance. This type
of iterative constructs are common in e-mail: it is the citation
of a message (quoting of other messages in another message). With regard to the cita-
tion, if a reference to the original message is needed then the referenc-
ing component (see below) can be utilised.14

14. Normally the citation comes from the replied-to message and therefore the message
ID of the original message would already be in the respondingTo component.

Figure 5.5: The grammar of FLC.

(1) <conversation> ::= <msg-st> <conversation> | <>
(2) <msg-st> ::= "msg(" <sender> "," <receiver> ","
<attitude> "," <content> "," [<context]
{"," <context)}> "," <message-id> ")"
(3) <context> ::= <pred-st> | <msg-st>
"and{[" <msg-st> "," <msg-st> ]")"
"or{[" <msg-st> "," <msg-st> ]")"
"isNot(" <content> ")"
"if(" <content> "," <content> ")"
"ifThen{[" <msg-st> "," <msg-st> ]")"
"ifThenElse{[" <msg-st> "," <msg-st> ]")"
"simpleUtterance(" <sender> "," <receiver> ","
<attitude> "," <content> ")"
(4) <sender> ::= <person-id>
(5) <receiver> ::= <person-id>
(6) <context> ::= "respondingTo(" <msg-id> ")"
"referencing(" <msg-id> | {<msg-id> ")"
"timeSent(" <time> ")"
"sendingMachine(" <machine-id> ")"
"taskName(" <string> ")"
"pendingTask(" <yes-no> ")"
"alsoSentTo(" <person-id> | {<person-id> ")"
"alsoAddressedTo(" <person-id> | {<person-id> ")"
(7) <pred-st> ::= <predicate> | "<arg> | "<arg> ")"
(8) <arg> ::= <person-id> | <message-id> | <document-id> | <time-pred> | <pred-st> | <priority> | <string>

Syntax:
A ::= B "A is defined as B"
A  B "A or B"
[A] "zero or one instance of A"
[A] "zero or more instances of A"
<x> "the non-terminal symbol A"
"abc" "the terminal symbol abc."
The other components in the `<content>` component—`and`, `or`, `isNot`, `iff`, `ifThen`, and `ifThenElse`—provide us with a mechanism for adding follow-up actions to messages. An example of the use of some of these components is given in the description of the APP message type in the next subsection.

Moving to the `<context>` part of the language (see line 6 in the figure), when a user replies to a message, the `respondingTo` component is used. The `referencing` component is used in similar cases, but it can on the other hand contain more than one message ID. These two components correspond to the use of the headers `In-Reply-To:` and `References:`, respectively, in e-mail (see Chapter 3). As another example of referencing, reminders (see vocabulary above) also need to reference an original message and subsequently the related conversation. The reference is contained in the `<content>` component as noted above. Notice that the reminder message is treated as a regular message statement and thus also contains the `<message-id>`. Hence, the reminder is assumed to be incorporated in the conversation when it is triggered. (Every message needs an ID in order to be referenced and managed.)

The `taskName` contains the name of the task; it corresponds roughly to the `Subject:` header. The `pendingTask` component is used for marking a message as belonging to an active task.

5.4.3 The Message Types in the Language

Here, we show how each of the message types presented earlier (see Table 5.1 on p. 86) can be represented by the language FLC that we have created (cf. Kimbrough & Moore 1997).

Read/Review/Comment (RRC) The RRC message type is a message with the illocutionary attitude of `request`:

\[
\text{msg}(\text{Sender}, \text{Receiver}, \text{request}, \\
\text{Content}, \text{Context}, \text{ID})
\] (5.1)

Here, the sender always desires some sort of response from the receiver. This is specified in the `Content` by using the different predicates—`commentOnItem`, `doable`, `reviewItem`, etc.—along with the time predicates found in the vocabulary of FLC (see Figure 5.4 on p. 94).

Appointment (APP) To represent the APP message type, we construct the following FLC structure, with the attitude of a `request`:
The content of the message includes the appointment predicate, where we define the time, date and place for the meeting. We can have several receivers, as described in conjunction with the grammar above.

As Kimbrough and Moore (1997) have pointed out, the strength, or the illocutionary force, of the meeting appointment directive can be expressed as a number from –5 to +5 as request(n), where –5 is pleading, +5 represents a direct ultimatum, and 0 is a polite request. Furthermore, if the sender wants an explanation when the request for a meeting is denied by the receiver, the content can be expressed as a conjunction with and (see line 3 in Figure 5.5) and the predicate reason (see line 13 in Figure 5.4).

The predicate absent (ibid.) in conjunction with <time-pred> can also be added to the response, for the purpose of listing dates and times when a meeting is not possible.

Dissemination of information (DOI) This type looks like this in FLC:

```plaintext
msg(Sender, Receiver, assert, interesting(Receiver, Item), Context, ID) (5.3)
```

In other words, the sender asserts that Item (the information in the message, which can be trivial or complex) is interesting to the receiver.

Staff action (SAC) The SAC type is based on the attitude request:

```plaintext
msg(Sender, Receiver, request, Content, Context, ID) (5.4)
```

The Content here describes to the receiver what action(s) she is supposed to take. If we want to automate the handling of SAC messages, we need a lexicon for representing the content of the message. Typical contents have to do with project status reporting, task assignment, and alteration of task priorities. As pointed out by Kimbrough and Moore (1997), we should strive to use basic—rather than derived—illocutionary attitudes, such as request for both APP (see above) and SAC messages. This means that when the system detects an APP or SAC message, it automatically assumes that the appropriate attitude is request. Also, in this case, there can be a limited set of permitted SAC message statements that the user can use. The information needed for the message is prompted from the user.

Query for information (QFI) The QFI message type is represented by the following FLC structure:
The Content is implementation-specific, i.e., the knowledge of what to do in response to a question is placed in the programs that use and process the FLC messages.

**Absence (ABS)** FLC represents the ABS message as an assertion with regard to the illocutionary attitude. The period of absence also has to be specified, so the message looks as follows:

\[
\text{msg}(\text{Sender}, \text{Receiver}, \text{assert}, \text{absent}(\text{From}, \text{Begin}, \text{End}), \text{Context}, \text{Content}, \text{ID}) \tag{5.6}
\]

In addition to the time period, defined by Begin and End, the absent predicate also takes the event, defined by From, as a parameter.

**Statement (STM)** The STM message type is modelled as a message with the illocutionary attitude of assert. In other words, the sender asserts that the statement she is making in the message is true. In FLC this is represented as follows:

\[
\text{msg}(\text{Sender}, \text{Receiver}, \text{assert}, \text{Content}, \text{Context}, \text{ID}) \tag{5.7}
\]

As in the case of the QFI message type (see above), the Content of an STM message is implementation-specific.

**Other (OTH)** The OTH message type is a message of the following format, with the attitude unknown:

\[
\text{msg}(\text{Sender}, \text{Receiver}, \text{unknown}, \text{Content}, \text{Context}, \text{ID}) \tag{5.8}
\]

### 5.5 Example: The scheduling-a-meeting scenario

We now revisit the scheduling-a-meeting scenario presented in Section 4.3 on p. 56. We extend the scenario by adding two more subtasks to it, in addition to the two previous ones (sending a request and scheduling the meeting). The new subtasks are: having the meeting, and writing and signing the meeting minutes.

To summarise, the task described in the scheduling-a-meeting scenario is to arrange and hold a meeting among a group of four people. There are four subtasks in the main task:
• compose and send a request for the meeting
• schedule the meeting
• have the meeting
• write and sign the minutes from the meeting.

The task is assumed finished when the meeting minutes have been signed. Moreover, we assume that there is a meeting room available for the task, but that the date and time for the meeting have yet to be decided.

We now show how the FLC language and the message types described earlier can be used to support task management in e-mail in this scenario. We first give an overview of the different message types used in the scenario, before we give language examples from the communication.

5.5.1 The message types in the scenario

The message types and the order of the messages exchanged in the scenario are shown in Figure 5.6 below. The number in front of the message type in the figure indicates the order.

![Diagram of message types](image)

**Figure 5.6:** The message types (and their order) in the updated scheduling-a-meeting scenario.

The purpose of the initial message in this scenario is characterised by the APP message type. The initiator (person 0) selects the meeting scheduling template containing the APP message type, composes a message and sends it off (step 1 in the order of messages implied in Figure 5.6). Three persons receive the request for a meeting. Through the use of the language the semantic context can be defined for the APP message type, and the
types of responses that can be expected. The request for an appointment requires the receivers to respond. The response to an APP message, according to our definition in Section 5.4.3 on p. 98, can be either a “no”—and possibly an explanation added to that—or a “yes” along with, if applicable, a list of times that suit (or do not suit) the receiver. In the figure STM (Statement) messages consisting of MDNs (see Chapter 3) are received automatically from persons 2 and 3 (step 2 in the figure), indicating that the message has been opened by the receivers in question. Also, person 2 sends a personally written response separately (the STM message type—step 3 in the figure). This response contains a citation from the original message received by person 2 from person 0, in addition to comments written by person 2. As can be seen in the figure, person 1 does not respond at all, and no automatic response (in the form of an MDN) is sent by her e-mail system to person 0, the initiator. However, the initiator of the task has programmed a reminder directly after sending the request for a meeting. This reminder times out (see step 4 in the figure) and the initiator chooses to make a telephone call to person 1 to get an immediate response (steps 5 and 6; the two continuous arrows between persons 0 and 1). After the telephone call person 1 composes and sends an e-mail message to person 0 (step 7 in the figure).

Now, person 3 happens to be the one responsible for writing the minutes from the meeting. When the meeting has been held, person 3 has to get the minutes signed by the group member responsible for checking the minutes, who happens to be person 2. The delivery of the minutes to person 2 and the signing of them are, in this scenario, both done manually using traditional mail—hence the continuous arrows between persons 2 and 3 (steps 8 and 9 in Figure 5.6).

5.5.2 THE LANGUAGE IN THE SCENARIO
We now offer examples of the use of the language. We show three specific cases from the scenario in detail. In the first case we show how the language works between the initiator (person 0), person 2, and person 3, who all have systems that understand the same language. In other words, they all have e-mail systems that implement FLC. This case corresponds to the communication between person 0 and persons 2 and 3 in the scenario (steps 1–4 in Figure 5.6). In the second case we give an example of how the language is used when one user has a system that implements FLC but the other one has not. In our scenario, this latter case corresponds to the communication between persons 0 and 1, where person 1 is not using FLC.
but person 0 is (steps 4–7 in Figure 5.6). The third and last case corresponds to the communication between persons 3 and 2 in the scenario (steps 8 and 9 in Figure 5.6). That is, the main communication takes place outside of the computerised system—a manual subtask (see Section 4.4.1 on p. 58)—but person 3 wants to enter an annotation about the event into the e-mail system.

The paragraph numbers below correspond to the numbers in Figure 5.6. To save space, some of the FLC components have been abbreviated, e.g., msg234 is used instead of message(234) as the value of the <message-id> component in the language.

Case 1  Both systems understand FLC.

(1) The initiator selects the template that has been created based on the APP message type and composes a request for a meeting. The message type enables the Conversation support module (see Figure 4.3 on p. 59) to format the message in FLC as follows:

\[
\text{msg(person(p0), person(p1), request, and[}
\begin{align*}
\text{msg(person(p0), person(p1), request, } \\
\text{appointment(at(time(2001, 7, 30, 13, 0, 0)), } \\
\text{at(time(2001, 7, 30, 15, 0, 0)), } \\
\text{string("conference room Eliten"), } \\
\text{person(p0), [person(p1), person(p2), person(p3)]}, \\
\text{[]}, \text{msg231a]), } \\
\text{msg(person(p0), person(p1), request, } \\
\text{commentOnItem(person(p1), string("This is the agenda: [...] Anything to add?")), } \\
\text{[]}, \text{msg231b}) \\
\end{align*}
\]

\text{[sendingMachine(obel10),} \\
\text{timeSent(2001, 7, 20, 18, 06),} \\
\text{taskName(string("Computer networks meeting")),} \\
\text{pendingTask(yes),} \\
\text{alsoSentTo([person(p2), person(p3)]),} \\
\text{msg231})
\]

This means that this is a message from person 0 (the initiator) to person 1 with the illocutionary attitude request. The message concerns both a request for an appointment and a request for commentOnItem—these two predicates are enclosed by the and component. The first part concerns the appointment, which will tentatively take place in “conference room Eliten” (the contents of the
string object) on July 30, 2001 from 13.00 to 15.00. The latter is defined by the time predicates at. The initiator of the meeting is person 0, which is deduced by the single person ID person(p0) listed last in the content. It is followed by the participating persons’ IDs within square brackets. In the second part of the and component, the initiator also wants a comment on the agenda and asks if there is anything to add. The current context follows, containing the ID of the sending machine, the date and time when the message was sent, and the task name (“Computer networks meeting”). The pendingTask component is set by each user; here, person 0 has set the value to “yes,” meaning she wants to track the messages belonging to the task. This is usually the default for new tasks. The alsoSentTo component indicates that the same message with the identical content and attitude (“the same token”) is also sent to persons 2 and 3—note that person 1 is already a receiver of the token on the first line. The message is identified by the number 231 in msg231.

(2) Person 2 receives the request for an appointment message from the initiator and opens it. Since person 2’s e-mail system handles MDNs, an automatic acknowledgement of person 2 opening the message is sent. The MDN message is formatted as a message of type STM using FLC and then sent to person 0 (cf. step 2 in Figure 5.6 on p. 101):

\[
\text{msg(person(p2), person(p0), assert,}
\]

\[
\text{opened(message(msg231)),}
\]

\[
\text{[sendingMachine(obel3),}
\]

\[
\text{timeSent(2001, 7, 20, 22, 55),}
\]

\[
\text{taskName(string("Computer networks meeting")),}
\]

\[
\text{pendingTask(yes),}
\]

\[
\text{respondingTo(msg231)],}
\]

\[
\text{msg234)} \quad (5.10)
\]

This message simply asserts to the initiator (person 0) that the previously sent message was opened by person 2. The context of the message contains the ID of person 1’s machine, the time the MDN was sent, and the ID of the message that the MDN is responding to. The number 234 is used to identify the message.

15. This is optional information and part of the implementation-specific parameters to the appointment predicate. Another option is to request a personal acknowledgement from the receivers (not shown here).
Person 3's system also responds with an MDN similarly formatted as in 2 above. The initiator's system, finally, receives the STM (MDN) messages from both person 2 and person 3, in addition to person 2's separate personal response. Person 0's (the initiator's) task context is updated accordingly.

(3) Returning to person 2's system, it subsequently categorises the message as an APP message. Optionally, the system can also recognise that an acknowledgement is requested (see step 1 above). In any case, person 2 decides to respond in person and confirm that she is available for the meeting. In addition, person 2 also wants to add an item to the agenda, at the same time citing the part of the original message dealing with the agenda. The message is formatted as follows:

```
msg(person(p2), person(p0), assert, and([
    msg(person(p2), person(p0), assert,
    available(person(p2),
    between(time(2001, 7, 30, 13, 0, 0),
    time(2001, 7, 30, 15, 0, 0)),
    [], msg236a),
    msg(person(p2), person(p0), assert,
    simpleUtterance(person(p0), person(p2), request,
    commentOnItem(person(p2), string("This is the agenda. [...] Anything to add?")),
    [referencing(msg231b), msg236b),
    msg(person(p2), person(p0), request,
    addItem(string("Yes. What about John's update of the lab series?"))
    [], msg236c),
]),
] sendingMachine(obel3),
timeSent(2001, 7, 20, 23, 14),
taskName(string("Computer networks meeting")),
pendingTask(yes),
respondingTo(msg231),
msg236) (5.11)
```

This message says that person 2 asserts (the illocutionary attitude) three things to person 0, enclosed in the and component. Firstly, she asserts that she is available (the predicate) between 13.00 and 15.00 on July 30, 2001, when the meeting is going to take place. Secondly, she asserts that she is citing what person 0 said to her in the previous
message, namely a request to comment on the agenda. This is done within the `simpleUtterance` component. To put it another way: “person 2 says that person 0 said `commentOnItem`.” There is also a reference to the original message in the `<context>` of this second part of the `and` component (cf. step 1 above). Thirdly and lastly, person 2 requests that an item be added to the meeting agenda. She does this by using the predicate `addItem`.

The message is concluded by the context, which contains the ID of the sending machine, the time that the message was sent, and the ID of the previous message to which this message is a reply (the `respondingTo` component). The current message is identified by the number 236.

(4) The initiator has previously programmed a reminder, so as to remind herself to check the meeting request status. This reminder is now triggered. The reminder consists of an STM message sent from the initiator to herself and looks as follows:

```plaintext
msg(person(p0), person(p0), assert,
remider(time(2001, 7, 26, 2, 0),
string("Check status of meeting!"),
priority(1)),
[sendingMachine(obel10),
timeSent(2001, 7, 26, 2, 0),
taskName(string("Computer networks meeting")),
pendingTask(yes),
referencing(msg231)],
msg251) (5.12)
```

In the reminder, the initiator asserts (the attitude) that the message is a reminder (the predicate). The time that the reminder was to be sent is July 26, 2001, at 2.00. The initiator has also provided a personal message concerning the topic of the reminder (“Check status of meeting!”). The reminder has been given the priority level of 1 (urgent).

The `timeSent` component in the context was added by the e-mail system when the reminder was finally transmitted. The original message (the request for a meeting which started the task) is referenced in the `referencing` component.

This ends case 1, which covered steps 1–4 in the scenario (see Figure 5.6 on p. 101). Since there are other pending requests for the same meeting (person 3—or possibly the automation in her system—has not reacted on the request at all yet) the status of the meeting is still tentative.
Case 2 Only one of the two systems understands FLC.

(1) Person 1’s system handles neither MDNs nor FLC. Now, let us assume that person 1, after getting a telephone call as a reminder from the initiator (steps 5 and 6 in Figure 5.6 on p. 101), decides to respond to the request for a meeting. She creates the message using the reply function of her e-mail system and sends the response to the initiator (step 7 in the figure).

(2) Person 0’s (the initiator’s) system now receives the message from person 1. The message is an ordinary Internet e-mail message and is shown in Figure 5.6 on p. 101.

(3) Person 0’s system now constructs the FLC structure that this message should contain, before the message is stored in the right conversation (thread) representing the scheduling-a-meeting task. The headers of the message are examined and parsed, and the FLC structure is filled in with the information found in the headers:

```
msg(person(p1), person(p0), unknown,
unknown(string("Hi: Thank you for the invitation.
I will attend the meeting! Cheers, Patrick"))},
```

```
Return-path: patla@portofix.ida.liu.se
Received: from obel3.ida.liu.se (obel3.ida.liu.se
[130.236.177.93])
by portofix.ida.liu.se (8.8.8/8.8.8) with SMTP id
JAA17390
for <juhta@ida.liu.se>; Sun, 22 Jul 2001 14:49:46 +0200
(MET DST)
Message-id: <200107221249.JAA17390@portofix.ida.liu.se>
To: Juha Takkinen <juhta@ida.liu.se>
Subject: Re: Computer networks meeting
In-reply-to: juhta’s message of Fri, 20 Jul 2001
18:06:51 +0200.
<200107201606.OAA06842@obel10.ida.liu.se>
Content-length: 72
Date: Sun, 22 Jul 2001 14:49:45 +0200
From: Patrick Lambrix <patla@ida.liu.se>

Hi:

Thank you for the invitation. I will attend the meeting!
Cheers,
Patrick
```

Figure 5.7: A typical reply to an e-mail message.
This FLC structure tells us that the message has been sent from person 1 to person 0. The illocutionary attitude is unknown, as is the predicate. The predicate unknown just lists the content of the body of the message as a string. The sending machine is identified, which is extracted from the last from entry in the Received: header—here, only one entry is available. The date and time for when the message was sent by person 1 are extracted from the Date: header. The message is marked as being a response to a previous message identified as number 231 (this number is in reality the ID between the angle brackets in the In-reply-to: header of the message). The message itself is given the identifier msg245, which corresponds to the Message-id: content. The ID in the respondingTo component actually makes it possible for person 0’s e-mail system to tentatively classify the message as belonging to the task named “Computer networks meeting.” (As it happens, person 1 composed the message by replying to the original request for a meeting.) The system’s suggestion for a classification is confirmed by person 0 and the classification is made permanent. Now each message in the initiator’s system concerning the meeting contains direct links to the relevant context via the FLC structure that has been put in the messages.

If person 1 had not used the reply function and had instead composed a new message, the In-reply-to: header would not have been available. This is a common situation, related to the handling of exceptions that we discussed earlier in this chapter and in Chapter 3. We discuss the implementation-specific details of this in Chapter 8.

The second case above we call the asymmetric case—only one system of the two involved understands FLC. In this case FLC is used as a data structure which makes task information readily accessible. The symmetric case is when both communicating systems use FLC, as in the first case above. Then FLC can support the control of who is in turn in the conversation, in addition to carrying links to relevant information.
Case 3  Information about a manual subtask

(1) Let us assume that person 3 has finished writing the minutes from the meeting and that person 2 has signed it. She now wishes to make a note about this in her personal task related to the meeting. She can do this in a number of ways, depending on personal taste. For simplicity’s sake, let us assume that she uses the e-mail system and sends herself a message stating the information that she needs to store. The message in FLC looks as follows:

\[
\begin{align*}
\text{msg}(\text{person}(p3), \text{person}(p3), \text{assert}, \\
\quad \text{note}(\text{person}(p3), \\
\quad \quad \text{time}(2001, 7, 31, 17, 0), \\
\quad \quad \text{string}("The minutes were signed by Peter."), \\
\quad \quad \text{document}(\text{doc22})), \\
\quad [\text{sendingMachine}(\text{obel5}), \\
\quad \text{timeSent}(2001, 7, 31, 17, 2), \\
\quad \text{taskName}(\text{string}("Computer networks meeting")), \\
\quad \text{pendingTask}(\text{yes}), \\
\quad \text{respondingTo}(\text{msg231})], \\
\text{msg634})
\end{align*}
\]

Here, person 3 has created the annotation by composing a message containing a note about the date and time when the minutes were signed, a string explaining the note, and a link to the minutes document itself. Person 3 has actually replied to the earlier received request for a meeting from person 0, which is identified by the \text{respondingTo} component. However, she has changed the \text{To:} and \text{Cc:} headers in the reply message to contain only her own address, and thus sent the message to herself. In this way the information that she wants can be entered into her e-mail system.\textsuperscript{16} The message ID \text{msg643} is relative to person 3’s machine.

As a side-note, since the task is now completed from person 3’s point of view, the \text{pendingTask} component above will be set to “no” when person 3 tells the system that the task is completed. One example of how a task is stopped is shown in Chapter 9.

This ends the language examples in the scheduling-a-meeting scenario.

\textsuperscript{16}  This common user behaviour was discussed in Chapter 3.
5.6 Discussion

5.6.1 FLC and Task Management

FLC can be used to generalise the tagged-message e-mail system as originally specified in RFC 822 (Crocker 1982). The context of a task can be specified by a series of declarations in the language FLC, which for example can be put in the header of an e-mail message. Furthermore, there is no specific order imposed on these declarations, which enhance the information in the e-mail message. The information thus declared can be used in an inferencing procedure.

FLC is entered into the system by the use of templates. For task management purposes FLC can be used to link messages and information sources—and also make it possible to express a rich variety of messages and conversations.

In addition to the simple addition of a task name to a message (the taskName in the <context>—see Figure 5.5 on p. 97), with FLC we can model reminders, the to-do list and the to-watch list. These are central aids in managing tasks. Firstly, reminders can be constructed using the reminder predicate as described in Section 5.4.1 on p. 93, in conjunction with the timeToSend component, which defines the delay for the sending of the reminder message. Secondly, the to-do list can be constructed by messages that contain the actionRequired predicate. This predicate adds a priority, deadline, and a note to a message representing an action that is required within a task. Moreover, the to-do list can also function as a reminder, but only in a passive way—the user must actively browse the to-do list so as to be reminded. Finally, the to-watch list, or the list of pending tasks, can be constructed by listing messages containing the pendingTask predicate.

The turn-taking of interaction partners is loosely represented by the common threads available in Internet e-mail. However, as stated earlier (see Chapter 3), a user does not always employ the reply function when she answers a message. Instead she may start composing a new message, relating the topic to that of the previous message. This breaks the chain of messages belonging to a thread, as defined by the Internet e-mail protocol. Other means, such as similarity matching and machine learning (a topic for the next chapter), are then needed to support the recognition of messages that should belong and be kept together. The FLC information can be used for this purpose. By using FLC to structure conversations a backup system is available for finding matching threads.
5.6.2 EPILOGUE

We have described a Formal Language for Conversations (FLC), FLC. The language can be used for describing message types and formalising conversations.

The language is simple but not too simple, and uses speech act theory (SAT) as a reference model. For example, we limit the number of attitudes to three, in addition to the “unknown” attitude. In addition, we have decided to initially implement only one attitude (assert, query, request, or unknown) per message. This is different from Kimbrough and Moore (1997), who model formal business transactions in their FLBC. We suggest that our approach suits the Internet e-mail domain better. One reason for this is that e-mail messages tend to be short in length, therefore allowing more than one attitude would be overdoing it. Furthermore, e-mail netiquette typically recommends one topic per message. This approach also facilitates simplicity in the user interface.

SAT is a general theory of communication, and thus potentially enables us to accommodate every possible message type in the e-mail system. We have (at least) three reasons for using a language such as FLC:

• we wish to store information related to a conversation together with the messages
• the information stored in the FLC structure can be used as a backup when the threading in e-mail fails (cf. Chapter 3)
• the FLC structure and the information contained therein can be used for message routing.

Since FLC facilitates storing of more relevant information together with the messages, this also makes it possible to do inferencing related to messaging.

Next, we discuss extensions to our conceptual model and implicitly also the language presented in this chapter.
In this chapter we present related work. We first review a representative selection of research message-based systems, which also include task management in one form or another. We conclude the chapter with a comparison of our approach with the approaches used in the systems presented.

6.1 Task-oriented restructuring of domain

There have been some studies of restructuring the application domain of e-mail. A task-oriented restructuring of the application domain of e-mail was presented by Fleming and Kilgour (1994). A number of general tasks that users perform using e-mail were identified, such as problem solving, negotiation, discussion, and request for and dissemination of information. Different types of messages were identified, inspired by Crowston, Malone and Lin (1988), which could be used to reinforce or suppress different characteristics of certain messages to make the presentation on the screen more readable.

A study of using e-mail for personal information management in LOTUS NOTES ("IBM/Lotus" 2001) was performed by Whittaker and Sidner (1996). The purpose was to propose technical solutions to e-mail overload (cf. Chapter 1). Whittaker and Sidner presented a list of suggestions for redesigning e-mail to fit the functions of filing and task management. These suggestions were in addition to the primary purpose of e-mail—enabling asynchronous communication—and included the following technical solutions:
• allow viewing by thread to reduce clutter in the inbox.\textsuperscript{17}
• use information retrieval (IR) techniques (Salton 1989) to cluster semantically related documents automatically and dynamically, so as to help the user find new categories to use for folders
• make it possible to mark particular inbox items as requiring action
• make it possible to program reminders that make action items re-appear as the deadline approaches.

\textsc{Lotus Notes} is a good representative of current technology—it is one of the most popular applications used by organisations (Wettemann 1998). However, the solutions suggested by Whittaker and Sidner need to be generalised to open systems such as Internet e-mail. We described a number of commercially available groupware products in Chapter 2.

With regard to research systems, we now continue with a review of related work in the area of message-based systems that support task management.

### 6.2 Different views of message-based task management systems

Here, we describe fourteen research systems. The systems are listed alphabetically with their authors in Table 6.1 below.

These message-based systems are often found referenced in research literature as examples of message-based systems including some mechanism to describe and support task management (cf. Rodden 1992; Bannon 1993). This is the criteria for including them in the overview that follows. Each system is treated in a separate section below.

<table>
<thead>
<tr>
<th>Section</th>
<th>Name of system</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.1</td>
<td>\textsc{Active Mail}</td>
<td>Goldberg, Safran, and Shapiro (1992)</td>
</tr>
<tr>
<td>6.2.2</td>
<td>\textsc{AtomicMail}</td>
<td>Borenstein (1992)</td>
</tr>
<tr>
<td>6.2.3</td>
<td>\textsc{Chaos}</td>
<td>De Cindio and colleagues (1988)</td>
</tr>
<tr>
<td>6.2.4</td>
<td>\textsc{Commodious}</td>
<td>Holm (1996)</td>
</tr>
<tr>
<td>6.2.5</td>
<td>\textsc{The Coordinator}</td>
<td>Winograd and Flores (1996)</td>
</tr>
<tr>
<td>6.2.6</td>
<td>\textsc{Information Lens/Oval}</td>
<td>Malone and colleagues (1987a)</td>
</tr>
</tbody>
</table>

\textsuperscript{17} This can also be used to reduce clutter in other folders.
6.2.1 **ACTIVE MAIL**

**ACTIVE MAIL** (Goldberg, Safran, & Shapiro 1992) uses e-mail to establish direct and persistent interactive connections between users and groupware applications. The connections themselves are not e-mail-based. **ACTIVE MAIL** is implemented under the LOGIX distributed programming environment, in a subset of the concurrent logic programming language FCP (see Goldberg, Safran, & Shapiro 1992). The user interface is based on the X WINDOW SYSTEM and OPENLOOK.

The functionality of **ACTIVE MAIL** is as follows. When an **ACTIVE MAIL** message is received, the recipient can interact with the sender, future recipients, and with remote, distributed multi-user applications such as shared editors with annotation capability. For example, *conversations* can be created when a message is received from an initiator. Here, a conversation is a list of the messages or documents that have been made available on a topic between participants. **ACTIVE MAIL** uses the notions of *publishing* and *sending*. Publishing a message or a document makes it available to all current participants. Sending a document to another user allows that user to become a participant. A contribution to a conversation can be made by publishing it into a shared conversation log window and sending the conversation to one or more receivers. The **ACTIVE MAIL** messages appear as entries in the receiving users’ folders.

**ACTIVE MAIL** extends ordinary e-mail with active messages, which contain facilities that allow communication and cooperation with other users. **ACTIVE MAIL** also retains all the features of e-mail. This makes it possible to smoothly migrate from simple use of ordinary passive e-mail to more advanced use of **ACTIVE MAIL**’s active messages. Among the groupware...
already realised within the ACTIVE MAIL framework are a collaborative writing facility and an interactive meeting scheduler. In principle ACTIVE MAIL can be used over the Internet, with multiple distributed servers.

### 6.2.2 AtomicMail

AtomicMail (Borenstein 1992) is based on the concept of computational mail, i.e., messages with embedded programs. It can be used to implement a CSCW application on top of e-mail. It represents an advanced way of embedding and sending programs in e-mail, for the automatic execution at the receiving side. With these programs a sender can, e.g., automatically create folders and install program files on a recipient’s computer, independent of what the recipient’s user interface looks like (Borenstein 1992).

AtomicMail is a language based on LISP, but because of security problems the LISP primitives for reading and writing have been removed. Furthermore, instead of the read function `read` some new and limited functions, such as `getstring` and `boolean` have been defined. The functions for handling files also have some restrictions for security reasons.

AtomicMail can be used to solve negotiation problems, e.g., when a group of people has to decide upon a common meeting time and place. An AtomicMail message can then be sent to each participant, which includes a program that presents the options (times and places) and enables each participant to click buttons in the e-mail message to reject the unwanted options. AtomicMail then collects all the answers and returns them to the original sender, and also summarises them to find a suitable time and place for the meeting. The meeting data can then be sent to each of the participants and subsequently inserted into their calendars. Note that the sender does not have to read the individual messages to get an overview of the result.

An implementation based on JAVA and the e-mail client EXMH, using the ideas discussed by Borenstein (1992), was developed by Danvind and Mattsson (1996). An e-mail system coupled with a web browser can be used to implement the ideas of computational mail by, e.g., using clickable URLs in the messages or embedded JAVA programs.

### 6.2.3 Chaos

Chaos, which is an acronym for Commitment Handling Active Office Systems, is considered as one of the first CSCW prototypes. It is both a prototype and a method for modelling cooperative activities, based on Petri nets and speech acts (De Cindio et al. 1986). Chaos was inspired by The Coordinator and XCP (see Sections 6.2.5 and 6.2.14, respectively).
CHAOS uses the Language Action Perspective (LAP; see Winograd & Flores 1986) as a conceptual framework. It is one of three prototypes for CSCW developed by the Cooperation Technologies Laboratory at the Department of Computer Science (DSI) at the University of Milan, Italy.\(^ {18}\) CHAOS-1 was implemented on a DEC VAX/750 under UNIX, and written in FRANZ-LISP and PEARL, an object-oriented programming tool available in the FRANZ-LISP environment. A later version was implemented in a SUN network using the QSL language.

In CHAOS the office is seen as a network of conversations, which includes possible future conversations and committed activities. The conversations are viewed as a backbone for supporting communication, activity execution, and role-playing. A conceptual description of CHAOS is shown in Figure 6.1. The Conversation Handler handles conversations by

\[\text{At each speech act:} \]

- **Knowledge Builder** observes the speech acts in a conversation and uses the information to update the Knowledge Base. It updates the commitments network associated with each speech act and the degree of competence of the involved participants, i.e., their experiences.

Cooperation in the office is made up of communication and coordination and can be fully characterised under the assumption that an office is a special linguistic game (cf. THE COORDINATOR above). This game is constituted by a set of rules defining the conversations possible within it. It is also continuously changing under the perturbation created by the speech

\[\text{Figure 6.1: The conceptual kernel of CHAOS (from De Cindio et al. 1986, p. 331).}\]

\(^ {18}\) The most recent prototype and further extension of LAP is the Milan Conversation Model (MCM) (De Michielis & Grasso 1994).
acts that the conversation members perform during the conversation. Within this context, the CHAOS prototype aims both at supporting the conversations and at improving coordination of the office activity.

### 6.2.4 COMMODOUS

COMMODOUS is a method for modelling “the usage of IT” (Holm 1996).\(^{19}\) The name is an abbreviation for “COMunication MODe-ling as an aid to Illustrate the Organizational Use of Software” (ibid.). Specifically, COMMODOUS is a method used to create an abstract model of communication in an organisation. Both discourses and conversations are modelled in the method. A discourse is a set of speech acts that are treated as logically belonging together, e.g., make order and send invoice. A conversation, on the other hand, is a more informal interaction. It is characterised by local control instead of global control, i.e., the participants decide what the topic is and who get to talk at the moment of the conversation. A discourse is globally controlled by means of the constituting rules of well-formed discourse. A task in COMMODOUS can consist of discourses, speech acts, actions, and other tasks. An action is a non-communicative activity performed by an agent in the organisation. The use of both discourses and conversations is motivated by the fact that there is a built-in contradiction between standardisation and flexibility in conjunction with the interaction that occurs in organisations. A task can be found anywhere in the spectrum from a globally controlled rigid discourse to a locally controlled free conversation. Holm (1996) advocates the use of Habermas’ version of speech act theory as a reference model for modelling social interactions.

### 6.2.5 THE COORDINATOR

THE COORDINATOR (Winograd & Flores 1986) uses speech act theory (SAT) to specify messages and their relationship in a conversation. Messages are classified according to the action they are meant to perform, i.e., the attitude that they have (assert, request, etc.—see Section 5.3.3 on p. 90).

THE COORDINATOR was a first step in exploring how to develop an electronic messaging system to support coordination processes in organisations at the individual level. See Section 5.2.3 on p. 81 for more discussion about The COORDINATOR. See also COMMODOUS above regarding similar use of speech act theory.

---

\(^{19}\) Parts of a prototype were developed using PROLOG and HYPERCARD.
Comparison with Other Approaches

Based on the ideas from The Coordinator, new platforms and programs for designing and executing workflows have been developed by Action Technologies (2001). One result is the Message Handling System (MHS) e-mail protocol, a standard used in computer networks and part of the NOVELL NETWORK (“Novell” 2001). The technology goes beyond the general support provided by The Coordinator to incorporate user-designed workflows that are created from the basic workflow types (requests, promises, offers, etc.) (cf. Ljungberg 1996).

6.2.6 Information Lens/Oval

Information Lens is an “intelligent tool for managing electronic messages” (Malone et al. 1987a; Malone, Lai, & Grant 1997). It helps people filter, sort, and prioritise messages. Information Lens is based on four key ideas:

- semi-structured message types based on frames or templates
- sets of IF–THEN rules for automating the processing of messages
- consistent display-oriented editors for handling frames and rules
- an incremental adoption path.

A hierarchy of semi-formal templates is used to structure messages (see Figure 6.2). The templates aid the user in structuring a message depending on the purpose of the message. More specific templates are placed further down in the hierarchy and inherit properties of the more general templates above. The structure is used in the selection, sorting, and prioritisation of received messages. Also, it can be used for generating automatic answers and to suggest different answers to a receiver. For example, a template for making a meeting appointment contains fields for time, place, speaker, and subject. It is also possible to define one’s own templates.

A group of people should all use the same message templates to benefit from classification rules based on the shared message structures. This is not always possible. Another, more complicated, solution is to translate all the different types of messages into a common language (Malone, Lai, & Grant 1997). A hybrid solution called “partially shared views” was described by Lee and Malone (1990), which combines the two approaches. It lets groups of people develop and use shared sets of message type definitions (“views”). A user can adopt views and thereby receive messages directly from other users of the same view. Messages belonging to non-adopted views are automatically translated into one of the message types a user has adopted. Non-adopted views that really are specialisations of already known views of the user are translated to the “nearest common ancestor” mes-
sage type (Malone, Lai, & Grant 1997). For example, a non-adopted view of type “Seminar Announcement” would be translated into the nearest “parent” type, such as “Meeting Announcement” (Lee & Malone 1990). The use of translation rules and object inheritance are both part of the general problem of converting from one data representation to another, which is critical to distributed systems that support heterogeneity.

The IF–THEN rules make it possible to automatically move messages into folders, remove messages, and to forward messages to other users. An example of the graphical rule editor in INFORMATION LENS is shown in Figure 6.3. The rules are defined via menus and fields that are filled in by the user. In the figure the user is defining a rule that moves messages containing “CISR Lunch” in the subject header to the folder “CISR Lunch.”
Some default values are also offered for each field, along with an explanation of the field and available alternatives.

The fourth key idea mentioned above, “incremental adoption path,” makes it possible for individual users to use their existing system with no change if they so desire. However, making only small changes can give a user immediate benefits, and groups of users who adopt the changes receive additional benefits beyond the individual benefits.

INFORMATION LENS later resulted in a completely new system called OBJECT LENS, which then became OVAL, a more general tool for supporting many kinds of cooperative work and information management applications. For example, OVAL can be used to implement Winograd’s THE COORDINATOR—see Section 6.2.5 above. OVAL was implemented as a proof-of-concept prototype in MACINTOSH COMMON LISP on networked APPLE MACINTOSH computers.
INFORMATION LENS is said to be one of the earliest systems that used agents, long before the term was coined.

6.2.7 KMAIL

Schwartz and Te’eni (2000) describe a system called KMAIL, originally named HYPERMAIL (Te’eni & Schwartz 1999), which uses organisational memory (OM) to contextualise electronic communication. The authors define an OM as consisting of a (semi-formal) organisational knowledge base and a (formal) set of meta-knowledge that can be applied to the knowledge base. The meta-knowledge used by KMAIL consists of user profile information and shared semantics information, which is stored in a relational database. The OM is assumed to be Internet-based, consisting of URL-addressable HTML pages. Likewise, the e-mail client is assumed to be HTML-enabled. Figure 6.4 shows an example of how an e-mail message is contextualised in KMAIL.

![Figure 6.4: A contextualised e-mail message in KMAIL (from Schwartz & Te’eni 2000).](image)
The creation of the OM itself is beyond the scope of KMAIL. KMAIL provides context to an e-mail message by using knowledge-based “memory-concept associations” (shared semantics of a given concept) and by determining appropriate memory items through the creation of OM views (ibid.). The goal is to achieve richer communications to decrease the probability of misunderstanding. Users that have the same view of a concept share the semantics of that concept in the OM. An OM view is similar to a logically filtered view in a relational database. The OM view takes an OM’s logical content and produces a filtered view suited to a given user in a certain situation.

6.2.8 MMS/FLBC

Message Management System (MMS) is the name of the general communication model developed by Moore and Kimbrough (1994). It is exemplified by a prototype developed for an army office environment. An MMS is generally defined by the authors as a tool for conducting business, which includes office work, transaction processing, and other commercial activities. Examples of activities that an MMS can support are the announcement of meetings, information about software bugs, planning lunch, and the coordination of shipments of goods between firms, among other things.

Included in the MMS model is the use of a Formal Language for Business Communication (FLBC) (cf. Section 5.2.4 on p. 83). MMS passes messages of high formality, inspired by electronic data interchange (EDI) systems. FLBC is a highly expressive language for formal messages. It can express what the message does, what the message is about, and essential elements of this context (Moore & Kimbrough 1994).

A typical MMS using FLBC is shown in Figure 6.5. The application does not directly communicate with other applications but uses the MMS

![Figure 6.5](image.png)
for this process. FLBC is used in the communication. Rule-based inference engines (represented by the knowledge base, KB, in the figure) are used to act intelligently on messages, and the messages themselves are stored in a database management system (DBMS). Moore and Kimbrough (1994) compare the expressiveness provided by FLBC with the movable type in the printing industry. The movable type made printing feasible for more books by reducing the printing cost of any particular book. In the same way FLBC provides a small number of reusable components that can be combined in many different ways. If needed, new concepts can be expressed by integrating new predicates into FLBC, which can be done without disruption.

An application using MMS/FLBC has been built for the army office environment. The system was implemented using standard MACINTOSH menus and dialogue boxes and QUINTUS/LPA MACPROLOG for the inferencing. The dialogues are used for handling the complexity—there are many possible combinations of language components. The army office environment provided some advantages that simplified inferences: there are clear lines of authority, and the rank and relationship of the communicating parties involved carries a lot of information in themselves (Kimbrough & Moore 1997). The MMS/FLBC interface is shown in Figure 6.6.

![Figure 6.6: Constructing a message in MMS/FLBC (from Kimbrough & Moore 1997).]
It is also possible to use FLBC as an agent communication language (ACL), in the same vein as the more well-known agent languages KQML/KIF (Finin 2001)—and the more generic initiative FIPA (“FIPA” 2001).

### 6.2.9 RAP and CAP

Bocionek and Sassin (1993) have in their research project investigated how so-called dialogue-based learning and programming-by-examples (PbE) may be used in e-mail tools. They have built an experimental system called RAP (Room reservation APprentice) that focuses on analysing incoming and outgoing e-mail messages that contain bookings of meeting rooms. RAP uses some opening examples and elucidating questions for learning a finite-state machine (FSM) It is then able to reproduce the procedure of booking a meeting room via e-mail the next time. The clear advantage of using PbE is that the user, through the higher level of abstraction, does not need to bother about all the details of a programming language.

One of the assumptions for using RAP is that the messages are reasonably structured, i.e., the messages are built using templates or certain keywords. No natural language processing is used. Instead, predefined key phrases (meaningful sentences) and keywords are used for filtering.

A message created by RAP, with a request for an available meeting room, is shown in Figure 6.7. The key phrase is need a lecture room.

From this phrase RAP can determine the type of the message, i.e., if RAP

To: main.office@cs
From: jean@cs
Subject: Need a room on dec-24-1992, 2:30pm RAP-Requestxx123

I need a lecture room.
Please let me know if there is one available as follows.

Date: dec-24-1992
Time: 2:30 pm
Duration: 90 min
Seats: 30 - 50
Speaker: M. Sassin, MIT

You can make your reply computer-understandable by beginning your message with one of the following words:

No or room number

**Figure 6.7:** A request for a lecture room created by RAP (from Bocionek & Sassin 1993, p. 16).
"knows" the phrase. The message in Figure 6.7 is of the type **room-request**. If RAP does not know the type of the message, it initiates a dialogue with the user: RAP asks for the type of the message and the key phrase in it. The message types used are **room-request, positive-answer, negative-answer, alternative-proposal and other**.

A more specialised version of RAP for a different domain has also been developed: a Calendar APprentice (CAP) (Bocionek & Sassin 1993). Mitchell and colleagues (1994) describe CAP II, an extension of CAP with facilities to support autonomous negotiation of meeting dates via e-mail. A collection of CAP II agents forms a system of communicating FSMs that interact with each other. CAP II is a calendar manager that in a similar way to RAP learns a user’s scheduling preferences from experience. CAP II learns rules that enable it to suggest meeting durations, locations, times, and dates for a user’s meetings. The rules are generated using a decision tree algorithm (ID3 and backpropagation; see Mitchell 1997), and they are intelligible to humans, written as IF–THEN rules. The advice (how long a meeting, where and when) can be accepted or overridden by the user.

### 6.2.10 SAMPO

SAMPO (Speech-Act-based office Modelling aPprOach20) is the name for a methodology used to model office communication and the effects of this communication on commitments (Auramäki, Lehtinen, & Lyytinen 1988; Lehtinen & Lyytinen 1986).21 With regard to this aspect SAMPO is similar to COMMODOUS (see Section 6.2.4 above).

SAMPO is based on The Language Action Perspective (LAP), which has its roots in speech act theory (SAT) (Austin 1962; Searle 1969) and communication action theory (Habermas 1984).22 (See also Flores and Ludlow 1980.) LAP is a theoretical orientation for studying modelling, design, implementation and usage of information systems in organisational contexts. It is based on an action view on language and communication, emphasising what people do while communicating.23

---

20. Another explanation to the acronym can also be found, namely Speech-Act-based information analysis Methodology with comPuter-aided tOols (Lehtinen & Lyytinen 1986).

21. Sampo is also the name of a magic mill in the Finnish national epic Kalevala. The mill ensures unending wealth for its owner.

22. For a comparison of Austin and Searle’s speech acts and Habermas’ communicative action, see Dietz and Widdershoven (1991), and also Eriksson (1999).

23. LAP workshops have been held annually since 1996.
SAMPO can be used as a basis for comparing office systems to one another. The communication model is designed to limit the number of possible interpretations of a message by a receiver, which leaves little room for inferencing.

6.2.11 Task Manager

The Task Manager (also called the EuroCOOP Task Manager) is an example of a simple but powerful and generic separate tool for the management of distributed work (Kreifelts, Hinrichs, & Woetzel 1993). It was designed to encourage the organisation of tasks and cooperative work by the end-users themselves, with minimal initial formalisation—the only mandatory attribute of a task is its title. The Task Manager is implemented on top of a toolkit written in C++ and networked as a distributed client/server system on a Windows/UNIX platform.  

The Task Manager is modelled around the idea of *shared to-do lists* (see Figure 6.8 on p. 128), i.e., the focus is on sharing workspaces. The figure shows one specific user’s shared tasks. Users can organise cooperative tasks and monitor their progress, share documents and services, and exchange messages during task performance. A network-like representation is used to describe tasks. Users can create dependencies between tasks, and also between tasks and documents. Users can also set and modify attributes (state, deadline, etc.), and add, modify, and remove documents. The system enables self-organisation of the cooperative work by the users themselves. This is accomplished by the constant distribution of information about tasks by the system, which makes resources available across networks, keeps data up-to-date, and resolves conflicts of synchronisation. The system guarantees a consistent view of tasks for each user.

24. The Task Manager originated in the DOMINO office procedure system (Kreifelts et al. 1991). A procedure in DOMINO equals a task in the office. A protocol called CoPlanX, which is based on speech act theory, is used to structure the exchange of messages in a task. DOMINO suffered from a lack of integration of other tools—and informal communication—into the system. For example, DOMINO messages were treated separately from ordinary e-mail, i.e., the user had to switch tools to send an e-mail message concerning an office procedure (Kreifelts et al. 1991).

Structured messages and the automation of office procedures are central in the messaging system described by Tsichritzis and colleagues (1982). The system was created as a direct response to the (faulty) use of a traditional (tagged-message) e-mail system. It is considered to be one of the first prototypes of a multimedia office filing system.

In Tsichritzis’ system a user can file, retrieve, and locate messages, query and obtain data present in messages. Furthermore, the user can specify procedures (by specifying patterns in precondition sketches) that coordinate messages and act (the sketches are triggered) only when a related set of messages has been assembled. Also, procedures can be created that modify
and create messages, file messages in dossiers, and automatically forward received messages to other stations according to their contents.

Tsichritzis’ system manages messages as typed objects which can be sent from station to station. Any message structure can be exploited to find messages and to query the messages for information. Queries are specified similarly to preconditions and actions in the system: by example, using sketches (templates). A collection of sketches describes what is to be done, rather than how to do something. A precondition results in the system continuously looking for the specified messages in order to trigger a procedure. An action describes to the system what it should do with the messages. The specification facility is provided on top of a passive form-processing system called OFS (Hogg, Nierstrasz, & Tsichritzis 1985), which in turn provides an interface to MRS, a small relational database system. Both of these systems were written in C, within a UNIX operating system. The language used to formalise the messages is similar in (limited) expressiveness to EDI.

6.2.13 Woo and Chang’s system

Woo and Chang (1992) have implemented a set of communication tools for solving negotiation problems. They use a theory based on SAT which is defined by Ballmer and Brennenstuhl (1981). Here, every allowable speech act at each step in a conversation is laid out. Compared to the messages in INFORMATION LENS (see Section 6.2.6 on p. 119), the messages are not completely formal but they are more structured. The system has been implemented using an e-mail tool-kit called STRUDEL (Shepherd, Mayer, & Kuchinsky 1990), which provides facilities to manage e-mail-based conversations. STRUDEL runs on UNIX, using the X WINDOW SYSTEM and Internet e-mail.

Woo and Chung do not make use of inference in their system, although it has a theoretical basis.

6.2.14 XCP

XCP is a very early attempt, developed at Digital Equipment Corporation (DEC), for project management and general coordination of work (Sluizer & Cashman 1984).26 It was designed to assist an organisation in implementing and maintaining its procedures. The implementation was made in VAX LISP on a VAX 11/785 under the VMS operating system.

26. As a side-note, DEC was in 1998 sold to Compaq Computer Corporation.
In contrast to the Coordinator described in Section 6.2.5, which was explicitly developed for intentional speech acts, XCP focused on facilitating linguistic action among humans. This is a goal similar to the goal of CHAOS (see Section 6.2.3 on p. 116). The XCP concept person is a human being. An actor is some person who has assumed some role, e.g., project leader, software engineer, lecturer, or writer. A document is the symbolic representation of some paper form.

XCP assumes that people follow procedures when they do office work, i.e., XCP encourages an organisation to clearly define formal procedural obligations and relationships in the form of protocols.

### 6.3 Discussion and summary

In Table 6.1 below we give a summary of the related (research) work that we presented in this chapter, and offer a comparison with our system. The comparison is based on the key features present in each system. The symbol “■” in the table means that the feature is present. The “n/a” means that the feature is not applicable to the system in question. A symbol “❑” in a column means that the system in question does not fully support the feature or function. The meaning of each column in the table is as follows:

<table>
<thead>
<tr>
<th>Language</th>
<th>Does the system use a language such as SAT to structure messaging? In addition, this feature implies that the exchange of messages can be logged and subsequently analysed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Templates</td>
<td>Are templates used as a means for giving structure to messages?</td>
</tr>
<tr>
<td>Openness</td>
<td>Is the system open to third-party software?</td>
</tr>
<tr>
<td>Agents</td>
<td>Does the system use agent technology?</td>
</tr>
<tr>
<td>OM</td>
<td>Does the system incorporate the use of organisational memory (OM), especially other electronically available sources?</td>
</tr>
<tr>
<td>Social</td>
<td>Does the system (still) allow social interaction, i.e., informal communication?</td>
</tr>
</tbody>
</table>

How does then our approach relate to the systems and models described above—and what news does it bring us? As described above, KMAIL enhances e-mail by including information from the OM, in order to reduce
### Table 6.1: Summary of related research systems

<table>
<thead>
<tr>
<th>Research system</th>
<th>Language</th>
<th>Templates</th>
<th>Openness</th>
<th>Agents</th>
<th>OM</th>
<th>Social</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>ACTIVE MAIL</strong></td>
<td></td>
<td></td>
<td></td>
<td>n/a(^a)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. <strong>ATOMICMAIL</strong></td>
<td></td>
<td>■</td>
<td>■</td>
<td></td>
<td>■</td>
<td></td>
</tr>
<tr>
<td>3. <strong>CHAOS</strong></td>
<td></td>
<td>■</td>
<td>■</td>
<td></td>
<td>■</td>
<td></td>
</tr>
<tr>
<td>4. <strong>COMMODOUS</strong></td>
<td></td>
<td></td>
<td>n/a(^b)</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>5. <strong>THE COORDINATOR</strong></td>
<td></td>
<td>■</td>
<td>■</td>
<td></td>
<td>■</td>
<td></td>
</tr>
<tr>
<td>6. <strong>INFORMATION LENS/Oval</strong></td>
<td></td>
<td>■</td>
<td>■</td>
<td></td>
<td>■</td>
<td></td>
</tr>
<tr>
<td>7. <strong>kMAIL</strong></td>
<td></td>
<td>■</td>
<td>■</td>
<td></td>
<td>■</td>
<td></td>
</tr>
<tr>
<td>8. <strong>MMS/FLBC</strong></td>
<td></td>
<td>■</td>
<td>■</td>
<td></td>
<td>■</td>
<td></td>
</tr>
<tr>
<td>9. <strong>RAP/CAP</strong></td>
<td></td>
<td>■</td>
<td></td>
<td>n/a(^c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. <strong>SAMPO</strong></td>
<td></td>
<td>n/a(^d)</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>11. <strong>TASK MANAGER</strong></td>
<td></td>
<td>■</td>
<td>■</td>
<td>n/a(^e)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. Tsichritzis’ system</td>
<td></td>
<td>■</td>
<td>■</td>
<td></td>
<td>■</td>
<td></td>
</tr>
<tr>
<td>13. Woo and Chang’s system</td>
<td></td>
<td>■</td>
<td>■</td>
<td>n/a</td>
<td>■</td>
<td></td>
</tr>
<tr>
<td>14. <strong>XCP</strong></td>
<td></td>
<td>■</td>
<td></td>
<td></td>
<td>■</td>
<td></td>
</tr>
<tr>
<td><strong>Our system</strong></td>
<td></td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td></td>
</tr>
</tbody>
</table>

---

a. The concept of **ACTIVE MAIL** is to piggyback on ordinary e-mail and provide an information-sharing environment to a user, with the possibility to work in realtime.

b. **ATOMICMAIL** uses **LISP** to create active e-mail messages, but the interaction itself is in the form ordinary e-mail messages.

c. As mentioned earlier, **COMMODOUS** is actually a methodology to model office work. It is included here for the sake of completeness, since it is based on SAT.

d. **kMAIL** parses the messages so as to identify concepts, and it also has OM views, which structure messages.

e. The CAP II interface is built on **GNU EMACS**, and communicates with an underlying **LISP** process, which provides CAP II’s learning and inferencing capabilities.

f. **SAMPO** is actually a methodology to model office work—it is included here for the sake of completeness, since it is based on speech act theory.

g. The electronic environment is created and contained in the tool itself.
CHAPTER 6

132

misunderstandings between the communicating parties. In our approach, we too include information from the OM. Furthermore, we also say something about the messages, which we do by using the language FLC. In KMAIL this “aboutness” is expressed more implicitly, via the use of hyperlinks to OM sources. Finally, we treat e-mail as a part of the OM, whereas KMAIL treats e-mail as a launching point into the OM (cf. Te’eni & Schwartz 1999).

Similarly to many of the systems listed above, we explicitly state in our system that we handle conversations, and not only messages. For example, in both CHAOS and XCP there exists the notion of a conversation, which in both cases is a sequence of communicative events between two or more persons. In both THE COORDINATOR and CHAOS the user is guided by the user interface to perform speech acts that are consistent with the set of available speech acts in the current conversation status. The only conversation allowed is the one based strictly on speech acts. A conversation in our system, on the other hand, is not restricted to speech acts. The unit of a conversation in our system is more general and is used to group any messages together that belong to a task. Speech acts are used as a reference model, so as to determine the general nature of a conversation. The electronic environment includes sources available electronically to the user, and these are used in the handling of conversations. Lastly, by including the manual environment in the model we can see traces of the user’s interaction with the manual environment in the electronic environment, which in turn can affect the handling of conversations.

CHAOS is actually more than an e-mail system supporting message exchange and retrieval, and more than a conversation handler, such as THE COORDINATOR. CHAOS—and also XCP—demonstrates the versatility of using LAP to model especially cooperative work. The theoretical

h. Messages are treated as attributes to tasks, which means that they only exist in the context of tasks.

i. The OFS in Tsichritzis’ system distinguishes between form types, form blanks (the template), and form instances (a filled-in template). A form type is the specification of a form blank and a set of field types.

j. Tailoring can be made using WINTERP, which provides an interactive object-oriented interface to the OSP/MOTIF UI TOOLKIT, using XLISP’s light-weight interpreter and object system (Shepherd, Mayer, & Kuchinsky 1990).

k. This can be accomplished by using notifications, a special kind of action item in STRUDEL to inform of events in an external tool.

l. XCP makes use of coordination centres, which are related to the role of a user and which contain all the material needed for the user to perform a role.
Comparison with Other Approaches

framework includes the development of active theories of coordination to facilitate interactions between humans. This is an objective similar to the one of SAMPO and COMMODOIOUS, but on a higher level. These two represent approaches that strive to create an abstract model of the communication, e.g., in an office. COMMODOIOUS is interesting in that it also tries to incorporate social interaction and not just communication directly relevant to the execution of a task. Communication described in both COMMODOIOUS and SAMPO tend to quickly become highly complex networks of speech acts, even for the simplest conversation. The conversation structures are restrictive and allow only a small range of possible messages at any particular time.

ATOMICMAIL demonstrates how a message can be made more active in that it can be constructed to contain code that, for example, automatically creates a filtering rule in the receiver’s e-mail client. All of this must of course be done with the receiver’s permission since there are security integrity issues involved. An implementation in JAVA exists of the concept (Danvind & Mattsson 1996). However, we believe that it should be up to the user herself, or “her” software, to construct the filtering rules in this case. This is because the rules need to be put into the context of the user, as defined by the electronic environment.

The TASK MANAGER is interesting in that it is organised around shared to-do lists, and that e-mail messaging has been added to the system as an attribute to tasks. However, it is not clear how a task can be saved for later perusal, e.g., for modification, storage and later re-use.

Regarding INFORMATION LENS, the biggest advantage is that organisations are given the possibility to define their own templates and rules for better internal handling of e-mail messages. This may contribute to higher productivity, faster decisions, and, generally, clearer communication (see Malone et al. 1987a, p. 128). A method for sharing the templates used in INFORMATION LENS was described by Lee and Malone (1990). However, they did not address the so-called $m \times n$ argument (Padlipsky 1983) when they discussed the “partially shared views” (see Section 6.2.6 on p. 119). Given $m$ systems sending data to $n$ distinct systems, writing $m \times n$ translation rules is not an effective way of handling different types of systems (see Figure 6.9 below). Every time an additional system is added, another $m$ translation rules must be implemented. This is the classical problem which is solved by the use of a common protocol (the network shown to the right in Figure 6.9). In conclusion, peer-to-peer translation schemes such as the one Lee and Malone (1990) propose are undesirable from a
software engineering point of view. One of the disadvantages with INFORMATION LENS is that both the sender and the receiver must be using INFORMATION LENS for efficient communication. Moreover, it does not use a special language to describe the communication.

MMS/FLBC described in Section 6.2.8 on p. 123 is similar to THE COORDINATOR (see Section 6.2.5 on p. 118). However, compared to MMS/FLBC, THE COORDINATOR is more like a tagged-message system (see Section 5.2.3 on p. 81) because the content of a message in THE COORDINATOR is expressed separately from the SAT-based classification on the actions that the messages are meant to perform. FLBC, the language used in MMS (Kimbrough & Moore 1997), is designed to replace EDI. In order to do that, FLBC is highly formalised and structured, which involves replacing the English in e-mail messages with a theoretically sound formal language. Social interaction, i.e., ad hoc and spontaneous conversation, is thus difficult in MMS/FLBC. The same applies to Woo and Chung’s system described in Section 6.2.13. Our approach FLC is not designed to replace the messaging system in the way that FLBC is. FLC is designed as a support to the messaging system (e-mail) and the user, adding information in and about the messages. Specifically, FLC is designed to support tasks in the context of e-mail.

As in ACTIVE MAIL, the ordinary e-mail usage is extended in our system. However, compared to ACTIVE MAIL, our system strives to keep the original interface of the user’s e-mail client intact, so as to keep the learning threshold relatively low. We furthermore emphasise that the electronic environment in our system also comprises the message-handling system itself. Also, ACTIVE MAIL does not make use of any special language to describe the conversations that occur. In a similar manner, Tsichritzis’ sys-

Figure 6.9: The $m \times n$ argument for using common protocols.
Comparison with Other Approaches

tem (see Section 6.2.12) does not make use of a language to describe or store information about the communication. Tsichritzis’ system is specifically designed for document management using templates (sketches or forms), without any explicit description of the communication.

Although RAP and CAP are designed for very specific tasks (room reservation, and calendar appointment management, respectively), they show the feasibility of keyword-spotting rules and machine learning to construct agents that help the user. We also found this combination interesting to try in our prototype.

6.4 Conclusions

The problem of describing and controlling cooperation in asynchronous systems (cf. Chapter 2) is being addressed mainly by two research areas: speech act systems and office procedure systems (Roddin 1992). The latter approach relies on a procedural language, typically object-oriented, to specify the office tasks. This language is then used to coordinate cooperation within an office. However, research has shown that the procedural language alone in this case is unsatisfactory. Therefore AI technology has been introduced to support these systems, e.g., using an AI-based planner to execute actions and achieve goals described according to plans expressed in some script language (ibid.). The former approach, using speech act theory, is the one that we have opted for. We discussed this approach in Chapter 5. However, we also include AI techniques (described in the next part of the thesis), which makes our approach come closer to the latter approach.

This ends Part II of the thesis. We initiated a restructuring of the application domain of e-mail with the description of the conceptual model in Chapter 4. Thereafter we developed a language for describing conversations and supporting the management of tasks in e-mail. The review in this chapter listed some related work. In Part III coming next we present an implementation based on the conceptual model.
IMPLEMENTATION
Chapter 7
Design of an E-Mail-Based Task Management System

In this chapter we examine the steps towards an implementation of the conceptual model presented in Chapter 4. We focus the discussion on the following three aspects of the model (cf. Figure 4.3 on p. 59):

- **conversation support**: representing and structuring messages, as well as information about messages
- **sorting and prioritisation**: filing and accessing messages in folders, and also as part of conversations
- **the electronic environment**: harmonising the information available in personal and external sources, and also accessing and using it.

These three components represent the structures, mechanisms, and environment that are needed for supporting task management in our model. We start the chapter by discussing the design requirements that we have to take into account when implementing a prototype. Subsequently, for each aspect above we examine the requirements and implementation alternatives, and present and offer support for the choice that we have made. Since the prototype that we implement is only one possible implementation of the model, we conclude the chapter with a look at the general problem of how to integrate the ideas of the model with legacy systems.
7.1 The model and the e-mail domain

We want the e-mail client to continue to be a client with which the user is familiar. In other words, when we introduce our system it should mix well with the functionality of the e-mail client. For example, folders should still be represented in the same way, i.e., as collections of browsable message summaries in scroll windows.\(^\text{27}\)

When we designed the conceptual model, we discussed system services and functions (see Section 4.5.1 on p. 62). Appendix B lists the system services and system functions for an e-mail system that supports task management. A system’s technical structure forms a framework for the implementation of these system functions. The distinction between system functions, system services and the technical structure of e-mail is sometimes vague (cf. the discussion in Section 4.5.1 on p. 62). For example, the protocol for representing and transporting Internet e-mail (Klensin 2001; Resnick 2001; Freed & Borenstein 1996) defines part of the technical structure. Some of the e-mail protocol elements, such as the e-mail headers, are also system functions. Some of the elements are even visible in the user interface as extra headers. These extra headers are normally not needed for the e-mail users to understand the messages.

In Figure 7.1 on p. 141 we have extended the user interface that we showed in Figure 5.1 on p. 77. We have added the relations to the system services and functions discussed earlier, as well as the technical structure and delivery system of e-mail discussed above. The dotted box represents the user interface—the terms used here were discussed in Section 5.1.2 on p. 75. The arrows represent the supporting or realising function the concepts have on each other. For example, the system functions support the (function of the) system services. Some of the concepts support other concepts higher up in the hierarchy, as discussed above.

We now continue with the requirements, the analysis, and the design of the three aspects introduced in the beginning of this chapter: supporting conversations, sorting information, and harmonising information.

---

27. Browsing is a central strategy in accessing information (Marchionini 1995).
7.2 Requirements

7.2.1 Supporting Conversations

The conversation support is mainly about organising text and storing it for later retrieval. This requires a representation of (and structure for) the information contained in the (text) messages. In Chapter 5 we justified the use of a language, FLC, to structure and store information in and about messages. The question we now pose is: From where and when do we get the information to use with FLC? We also wish to know how FLC can be
introduced into the system, i.e., how FLC can be implemented in an exist-
ing e-mail client. Furthermore, we want to know how the function of con-
versation support can be introduced to the user. How can the information in
and about messages be captured and subsequently presented to the user,
and used by the user?

We note that the conversation support needs to be both internal and
eexternal. With “internal” we mean that the system needs to structure and
represent information concerning messages and conversations for internal
processing. The structuring needs to take place when the user is compos-
ing (or has composed) a message. The message can be a new message, a
reply to an old message, or a forwarding of a previously received message.
Externally the system needs to present the information in, and about, mes-
sages and conversations to the user in such a way that the user can “under-
stand what is going on.” This presentation requires preprocessing of the
messages, which includes parsing URLs, if any, in the messages and cre-
ating links to related (external) documents. The user should be able to
change between different views, e.g., a single message, a thread of mes-
sages (a conversation), or a folder. At the extreme, the same message can
be allowed to be presented differently depending on the user’s current sit-
uation, e.g., if the user is under light, normal, or heavy workload (cf.
Takkinen 1998a). Hence, some kind of presentation rules are needed.

Getting an overview of the available information is a key concept in our
model. To get an overview of current conversations, the user needs a means
of compacting or summarising the information requested and displayed.

The threading mechanism is also needed for effective inferencing—the
thread creates a relevant context for a message in which the system can do
its searches. With “inferencing” we mean the making of implicit informa-
tion explicit. For inferencing to be effective we need access to correct and
relevant information. If the messages belonging to a task are all collected
in one thread28—and the thread is intact—we have the right “core con-
text” and a good starting point for doing inferencing. We return to a dis-
cussion of threads in Section 7.2.2 on p. 143, when we examine the
requirements for the aspect of sorting information in our system.

Common to our requirements, discussed above, is that we want to be
able to combine, exchange, and present text (and possibly other media).
The text must be organised within some infrastructure.

28. This includes both incoming and outgoing messages.
7.2.2 Sorting messages

When a message is sorted it is filed in a folder, and subsequently in a thread. The sorting is assumed to be automated, e.g., by the use of filtering rules. However, we require the automation to be accessible to the user. That is, the user should always be aware of what is happening. The user can also manually sort messages. A message previously stored in one folder can be moved to another folder by the user. If automation of the sorting mechanism is employed, a manual refiling done by the user should update the rules used by the automation. The deletion of a message is similar to a refiling of a message: it is simply a refiling of the message to the trash (or corresponding) folder. The difference is, however, that the rules should not sort any incoming messages directly to the trash folder, i.e., the trash folder is not included in the rules. The reason for this is that we only allow the user to delete messages. Many people depend on getting e-mail reliably. Furthermore, most individuals (if not all) do not want their e-mail system to automatically delete e-mail without letting them read it first (see, e.g., Takkinen 1994). Also, you can lose some or all of your incoming e-mail if your automatic management of e-mail is not working correctly or is not giving you the right feedback. These facts need to be considered in the implementation.

Specifically, if an ML algorithm is used, we require the rules produced to be visible and accessible to the user, so that the user can understand how the system reasons. That is, when a classification has occurred, the user should be able to understand why the system did the classification.

A task should be easy to start, e.g., by the user simply creating or selecting a message representing the first message in a (future) thread. The selected message can be a newly arrived one, which prompted the user to initiate the task at her side. It can also be an old message stored in one of the non-task folders. A task should be equally easy to end by the user, e.g., by selecting a message in the task and at the same time indicating that the corresponding task should be ended. This means that any filtering rules triggering on messages belonging to the task should also be deactivated. The finished task should then be moved to a permanent storage, in one of the folders specified by the user—if the task and its messages are not already in an appropriate folder, that is. The messages in a task should stay together, and the rules should be updated accordingly.

Finally, a ticker should check the deadlines of reminders defined by the user and send messages to remind the user about tasks.
7.2.3 HARMONISING INFORMATION

Here, we have the problem of how the information in the electronic environment can be harmonised and made operationally available as part of our system. A large part of the harmonisation is to standardise the information available from a limited set of sources, and make it available for processing in the context of a task.

In other words, the sources in the electronic environment are heterogeneous and distributed. For a simple example, dates and times are often expressed in different formats, depending on the application used. Figure 7.2 shows the different formats used by a calendar manager application and Internet e-mail, respectively.

<table>
<thead>
<tr>
<th>The calendar manager (dtcm) in CDE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appointments for Monday, 2001 July 23:</td>
</tr>
<tr>
<td>57) 1315-1500 Teaching meeting, Betinget</td>
</tr>
<tr>
<td>Internet e-mail:</td>
</tr>
<tr>
<td>Date: Mon, 23 July 2001 13:15:45 +0100</td>
</tr>
<tr>
<td>Subject: Re: Teaching meeting</td>
</tr>
</tbody>
</table>

**Figure 7.2:** Two examples of different date and time formats.

We discussed the electronic environment and its sources in Section 4.4.2 on p. 61. We divided the sources into personal sources and external sources. Two examples of external sources are the database of employees and the project database. The database of employees is a centrally administered database with information about the employees’ names, addresses, affiliations, etc. in an organisation. It is usually administered by a very small group of people and made available over the network to a department in an organisation. With a project database, the user has access to external information about current projects. Information in the project database can be updated and modified by several key project members, such as the project manager and the project administrator.

The distributed nature of sources, and the heterogeneous information available in them, calls for a special kind of architecture. It should be possible to automatically coordinate applications and sources with the e-mail client, but still keep the simplicity of e-mail use (composing and sending messages, reading, deleting, and storing messages, etc.).
7.3 Analysis

7.3.1 Supporting Conversations

The unit of conversations, introduced in the model in Chapter 4, can be used to reduce the amount of information presented to the user. This helps her to get an overview of the relevant information. In other words, messages have to be clustered together into threads. This reduces the amount of information displayed on the screen. This helps the user to get an overview of the information presented in the system. The thread is used to preserve the task context and to keep the relevant task information linked together. This leads us to the conclusion that we need a stable (robust) threading mechanism, i.e., a mechanism that collects messages that somehow belong together into threads.

Since we need to structure the information in a message, and also relate the message to other messages in a conversation, we can use the unit of a thread to traverse relevant messages and access relevant information. In other words, by reading the latest, single message stored in a thread we can find all the relevant information pertaining to people involved, documents distributed, web pages referenced, etc. Inferencing can then be done on this information, i.e., new information can be extracted which is not explicitly available in the messages. For example, we can infer who have not responded to a query.

With regard to organising text within an infrastructure, a common technique is to tag it, i.e., to mark parts of the text that need to be recognised and provide extra information about the text, metadata. The general standard for tagging text on the Internet is the eXtensible Markup Language (XML) from the World Wide Web Consortium (“W3C” 2001). We could also use a standard for representing metadata only, such as the Dublin Core (“Draft Standard Z39.85” 2001). However, XML also includes a dynamic component which we require, e.g., when presenting the text in different formats. XML, along with Document Type Definitions (DTDs; see Bradley 2000), is a promising technique for structuring and transporting messages. It does not contain the semantics of the communication, which we instead provide via the use of the language FLC. XML is thus used for encoding and carrying this language.

By coding the FLC structure in XML the information about the task is made readily available for processing. By using FLC we can include the attitude (force) of the message (see Section 5.3.3 on p. 90) in the task information. This makes it possible to infer some more information that is val-
uable when processing the message. For example, if the attitude is request, then the sender’s processing software could be configured to automatically look for responses to the message.

Regarding presentation rules for presenting the same information differently depending on the situation, we note that the cost of such flexible representations of information is in the various mechanisms for controlling the different representations. The mechanisms—usually paging, scrolling, and jumping—require the user to develop new strategies for manipulating the physical structure of the information, e.g., the length of a message or multiple windows on the screen (Marchionini 1995, p. 44).

7.3.2 SORTING MESSAGES
As described in Chapter 4, according to the conceptual model, the sorting of messages consists of two different stages separate from each other. In the first stage the incoming, new messages are sorted into either a conversation (a thread in a folder representing a pending task) or a non-task folder. This classification of messages corresponds to the use of automatic sorting in current e-mail software (cf. Chapter 3). The relevant conversation (thread) can be found in either a special folder containing all pending tasks, e.g., a Tasks folder, or a “normal” folder where some of the messages are grouped in conversations. In the second stage the new messages classified as belonging to an ongoing conversation are prioritised.

We see two alternatives to automating the sorting of messages into conversations (threads) and folders: either manually managed IF–THEN filtering rules, or the use of a machine learning (ML) algorithm to create the filtering rules. In its simplest form the rules in both cases are stored in a text file in one format or another. The manually managed rules are only a starting point for an automatic classification. For more short-term tasks the manually managed rules are not flexible enough—the filtering rules need to be updated when new tasks are started or finished. This updating of filtering rules is something a user will not be willing to do. Moreover, rarely do users create filtering rules at all (cf. Chapter 3). The starting of a task implies the starting of the automation of sorting messages into folders and threads.

As discussed in Chapter 3, the current threading mechanism used in e-mail is flawed. The classification of messages into relevant threads presupposes that there are threads available as targets for the classification mechanism.

In e-mail we can make use of two types of links between messages, in order to keep messages together in threads: structural links and semantic
links. The structural links are links already present in the messages, typically inserted into the header by the e-mail transfer system. Examples of structural links are the \texttt{In-reply-to:}, \texttt{Message-ID:}, and \texttt{Subject:} headers. The first two contain references to other messages in the form of message IDs, and the latter one in the form an identical text portion. Also, the components in the FLC-based information sent with a message consist of structural links, e.g., in the form of a task name. The semantic links can be created by making connections between documents based on the similarity of their information content. Typically, an e-mail reply message contains significant portions of text quoted from other messages in the same task. This makes matching based on similarity possible and feasible (see Lewis & Knowles 1997).

The filtering rules created by the ML algorithm can be used for both sorting and prioritisation messages. Prioritisation can be done by correlating information in incoming messages with information available in calendar events and also recently sent messages available from the outbox. Cross-referencing this information with information available in the address book and any web bookmarks of the user could also be possible. Marx and Schmandt (1996) describe a messaging system that infers message timeliness by considering calendar appointments, outgoing messages, and phone calls, and by correlating the information with an address book.

7.3.3 Harmonising information

Accessing different electronically available sources in a user’s environment and using it, e.g., for classification purposes, presents at least three complications (Genesereth, Keller & Duschka 1997):

- **distribution**: not every classification can depend on data from one single source—fragments of useful information need to be combined
- **heterogeneity**: different sources may be using different access languages and protocols—for example, different words may be referring to the same concept
- **instability**: existing sources may be changing the format of their data, or changing their content.

So-called intelligent agents (Jennings & Wooldridge 1998) are typically constructed to alleviate these complications. The typical metaphor used for agents is that of a personal assistant who collaborates with the user (Maes 1994). The idea is that agents should hide the complexity of difficult
information-handling tasks, and monitor events and procedures on the user’s behalf. Instead of always waiting for the user to take the initiative, the agents provide for a cooperative process in which both the user and the agents initiate communication, monitor events, and perform information-handling tasks.

For each type of information source—also called legacy component—a special program called *agent wrapper* or a *transducer* (mediator), depending on the characteristics of the legacy software, needs to be constructed. This special program translates between the “language” of the information source and the core system (Genesereth 1997) (see Figure 7.3).\(^\text{29}\) The wrapped legacy code (or the transducer) is introduced to the *facilitator*. The facilitator’s main function is to facilitate the communication between the agents. An agent must be registered with the facilitator to be allowed to function in the system. The communication between agents is done using an agent communication language (ACL). The ACL API (Application Program Interface) in Figure 7.3 forms the interface to the ACL (Genesereth 1997).\(^\text{30}\) In other words, a wrapper injects code into a program and makes it communicate in the ACL, while a transducer mediates between the existing program and the facilitator. A transducer does not require knowledge of the program other than its communication behaviour, which is especially useful if the code for the program is unavailable, for example.

In an especially dynamic environment where sources come up and down, and data is altered often, agents provide an infrastructure that can handle problems with information. As a further enhancement, the agents could be designed to roam the network and follow the user from computer to computer, thereby providing the user with the computing environment that she is accustomed to.

Finally, the information in the different sources of the electronic environment can be used for modelling a user’s interests more explicitly. How-

---

\(^\text{29}\) Another alternative is to rewrite the legacy software. This is a drastic approach, but it can enhance the efficiency of the software.

\(^\text{30}\) Typically, the ACL API supports the passing of ACL messages over the Internet, transparently employing several underlying transports such as TCP/IP, MIME multimedia e-mail, and HTTP, enabling agents to send and receive ACL messages via web browsers and servers. A popular ACL used for communication between agents and the facilitator is the Knowledge Query and Manipulation Language (KQML) (Finin 2001).
ever, this is beyond the scope of this thesis. See Kay (1994) for a discussion of user models in the context of a filtering system.

### 7.4 Design

#### 7.4.1 Supporting Conversations

Regarding capturing the information from the user, we have learned that we can use **templates** to introduce structure to a message (cf. Chapter 6). More specifically the templates introduce structure to the **message body**. By using FLC we can furthermore add structure to the message as related to other messages, i.e., a **whole conversation** can be given structure. This conversation can be stored as a thread. The structure enables the formalisation of the messages and the conversation. The idea is that when the user selects a template for a message she actually does a classification of the message. This classification is transferred to the receiver in the form of the FLC structure, which the receiver can make use of for supporting her own task management.

Unstructured tasks can, with FLC, be given a preliminary structure, a representation in the system that is searchable and possible to edit, optimise, generalise, and save for later (re)use. An example of this reuse is the programming of **reminders** associated with certain receivers of a message: After analysing a task, the user of the system can decide to associate a

Figure 7.3: Two approaches to integrating agents and legacy software (adapted from Genesereth 1997).
reminder function to certain e-mail addresses (representing receivers) that have shown tendencies to be late with answers to a request message.

The eXtensible Markup Language (2001), XML, provides us with an infrastructure for organizing text. It is also an open standard and is widely used, which are facts that fit our requirement of an open system (cf. Chapter 3). With XML, we can create semi-structured messages from templates, and use this structure in our message handling.

A summary of task-related information can be presented by first parsing the current task (conversation) and creating a separate window with all of the documents distributed so far in the task, the people involved, the URLs mentioned in the messages, etc. The window contains clickable links to the relevant messages and web pages.

### 7.4.2 Sorting Messages

To summarise, we can make use of three sources in order to keep a thread (and consequently a task) intact—and its messages linked together:

1. The FLC information stored together with each message
2. The message IDs stored in the headers of each message
3. The text in the message body.

Note that the two first items can be missing from a message. Firstly, we do not require another user to use FLC, and secondly, there exist e-mail systems that do not store message IDs in outgoing messages (see Chapter 3). Both of these two items are important for threading—they are needed for ordering the messages in a thread by task name (the FLC information), or by date and time (the message IDs). The third source is employed if either the In-reply-to: or References: header is not available. This can be the case when, e.g., the user, instead of making a reply to an old message, has composed a completely new message and thus broken the thread. The matching algorithm can then be used to find similarity between messages.

The ticker can be represented by a folder containing messages that are delayed until their deadlines are reached.

There are a number of ML approaches for creating filtering rules and automating e-mail classification—see Kay and McCreath (2001) and Takkinen (1997) for overviews. We suggest the use of an ML algorithm called RIPPER for rule learning to manage the filtering rules (Cohen 1996). The reason for this selection is that RIPPER produces rules in an understandable format, in contrast to Naive Bayes approaches where the user is una-
ble to decipher why a message has been placed in a particular folder. RIPPER is also fast. We describe RIPPER in Chapter 9.

7.4.3 Harmonising Information

INFOMASTER (Genesereth, Keller, & Duschka 1997) is a successful and generic information integration system. INFOMASTER has been in production use since 1996 for, among other things, meeting room scheduling. It provides integrated access to multiple distributed heterogeneous source on the Internet. We can base an agent infrastructure on the ideas from INFOMASTER. The core of INFOMASTER is the facilitator, which harmonises source heterogeneity by using rules and constraints to describe information sources and translations among these sources.31

Generally speaking, each type of box in our conceptual model (see Figure 4.3 on p. 59) can be said to represent one or more agents. The agent for the outgoing messages in the model creates direct and inverted index files that are handed to an indexing agent. The agent for calendar events also creates a direct index file and an inverted index file.32 The indexing agent merges the different indexes handed to it from the different electronic sources and also trims the vocabulary, possibly by stemming and removing words defined in a stopwords list.33

Consequently, an architecture for managing the electronic environment can be depicted as in Figure 7.4 on p. 152 (from Takkinen & Shahmehri 1999). Two common and heterogeneous information sources are shown in the figure: the calendar and the incoming/outgoing e-mail messages. The facilitator is the special system program described earlier. The information filtering (IF) agent classifies the incoming messages into tasks and folders stored in the personal database (PDB). The PDB represents a user’s collection of previously received and stored messages, not necessarily stored on one single computer. Since we have a requirement that sorting be automated by the use of an ML algorithm, there is also a

31. The internal content language used by INFOMASTER is the Knowledge Interchange Format (KIF) (Genesereth 1997; Genesereth & Fikes 1992), a LISP-like language used for representing first order logic expressions.

32. The direct index file is indexed on the documents, while the inverted index file is indexed on the keywords found in the documents. We use the term documents here to denote both individual e-mail messages, as well as attachments, calendar entries, etc. in the user’s electronic environment.

33. Stopwords can be words such as “it,” “the,” and “is,” which are considered too common to be useful in retrieval.
learning agent in the architecture. The learning agent combines the information from the sources in a user’s electronic environment to create and update classification rules for incoming messages as well as presentation rules for the views. The views are logical groupings of messages, such as a listing of all pending tasks or all messages sent by one person. The views are managed by the IR agent. One wrapper and one transducer (the shaded ovals) are also shown in the figure. The term index, handled by the indexing agent, contains the index to all electronic sources currently available in the system. The indexing agent also handles the vocabulary that is available to identify documents and use in classification rules.

We conclude that we conform to the idea of using a “weak” agent technology (Jennings & Wooldridge 1988) in which an agent is any software capable of sending and receiving agent messages following a predefined interaction semantics. In other words, the agents do not necessarily have to be “smart” or follow any particular theory of how to construct agents. The only commitment is to send messages conforming to a defined set of interactions (the ACL).

---

34. This includes the outgoing messages and any refilings of messages done by the user, in addition to the “standard” sources, such as the calendar and the address book.
7.5 Integrating our system with a legacy system

We now tentatively examine how the proposed implementation of the conceptual model can be integrated into a legacy system.\(^{35}\)

We first note that we base our support for task management on messages, which means that the mechanisms described here should degrade gracefully to any message-based system. Furthermore, by using standards in the implementation we ensure that other software understanding the same standards can interface with the implementation.

With regard to the three aspects discussed in this chapter, the integration process needs to consider the following.

7.5.1 Integrating supporting for conversations

Here, the integration requires access to the internal functions of the legacy software, if the ease-of-use of e-mail is to be maintained. This has to do with the creation of the FLC structure by the employment of templates. The templates need to be integrated with the user interface of the e-mail client, so that information such as the attitude of a message, as expressed using FLC (see Chapter 5), can be captured. The templates need to be created by the user, changed, and (re)displayed to the user, and particularly at the stage when the user is composing a message. Furthermore, the messages stored in the folders of the user need to be writable, so that FLC information in the stored messages can be added and (or) updated. The storage format of the e-mail messages needs to be considered.

Similarly, the user interface of the legacy software needs to be further modified if presentation rules are employed. That is, depending on the message type, current workload or the user, etc., different views of messages and folders can be used.

In order to support conversations where the delivery of a message and (or) the opening a message must be confirmed, the legacy system needs to implement the appropriate Internet standards. Message Disposition Notifications (MDNs) were discussed in Chapter 3, which is the Internet standard described in RFC 2298 to implement the confirmation of an opened message (Fajman 1998). The delivery of a message can be confirmed by

---

\(^{35}\) A more definitive examination is presented in the two chapters that follow (Chapters 9 and 10), where we show one specific implementation based on the ExMH e-mail client. The implementation is described partly from a developer’s viewpoint, and partly from a user’s viewpoint.
implementing Delivery Status Notifications (DSNs), another standard, described in RFC 1891 (Moore 1996).

7.5.2 Integrating Sorting of Messages

The ML algorithm that creates the filtering rules operates on the incoming and outgoing stream of messages in the e-mail system. This means that an integration of our system with a legacy system, in principle, means accessing, parsing, and acting on the messages after they are sent or before they are received by the e-mail client. However, there are changes that have to made to the user interface of the e-mail client. For example, an explanation of the classification of a message needs to be associated with the message—this is what we required in the design, i.e., the filtering rules should be visible and readable to the user.

The functions for starting and ending a task also require modifications made in the user interface. The legacy system needs to detect when the user has marked a message and issued the “start task” command, either implicitly by selecting a template, or explicitly, e.g., by pressing a key. When a task is finished the user issues the “stop task” command, which is normally issued explicitly as a command by the user.

The threading mechanism of the legacy software is affected, since the thread is a central concept in our model. The mechanism needs to be enhanced with the detection of FLC information (mainly the task name) and a matching algorithm for the text in the message body. We describe an algorithm more in detail in Chapter 9, Section 8.3.1 on p. 165.

Furthermore, threads need to be distinguished based on the task name, so that the user can view a list of pending tasks, or an overview of the messages in one certain task, for example. Also, in order to give the user the possibility to view a separate summary of task information, including persons involved in the task, documents, etc., further changes in the user interface are required (another command).

With regard to reminders, they can typically be programmed from the calendar software, e.g., the calendar entry can be set up to send an e-mail at the required point in time. In this case the integration is quite simple (none needed), otherwise the e-mail client needs to be updated with a reminder function.

If the legacy system uses HTML-based e-mail, a specific method for implementing a more robust threading can be followed—however, the threading still needs to be enhanced with similarity matching and FLC parsing. The method is described in a note published by W3C (Berman,
It explains what conventions should be followed for using HTML in e-mail to make threading useful. The note describes how properties of the text in the message and properties of the message itself can be tracked, such as who wrote what or what message a quoted excerpt is originally from.

### 7.5.3 Integrating Harmonising of Information

The sources in the electronic environment need to be able to publish their contents for use by the agent software. For example, the calendar software on many platforms can output the content to a text file—this can be accessed by the calendar agent for distribution to other agents. This is the transducer approach for integrating agents with legacy software, which we described in Section 7.3.3 on p. 147. If the internals of the source in the electronic environment are available for modifications, the wrapper approach can be used instead (ibid.).

When the agents have been programmed they need to be managed. The agents (sources) need to be introduced to the e-mail client, and the facilitator (see Section 7.3.3 on p. 147) made aware of the available agents in the system. This implies functions for defining, starting, and stopping the facilitator. Furthermore, the sources need to be defined to the e-mail client, registered with (unregistered from) the facilitator, and also connected to (disconnected from) the facilitator.

Messages can have annotations associated with them. The amount of integration needed so as to make the annotation function available to the user depends on the legacy software. If it is possible to modify messages after storing them in folders, then the annotation procedure is straightforward: The user opens the message and adds additional text. Otherwise a separate annotation software will probably have to be employed. In both cases the functionality in the user interface has to be modified so that it can update the FLC information in the message when an annotation is added.

The complete list of abstract and concrete services and functions that one needs to consider when integrating our system with a legacy system is available in Appendix B. The services and functions specific to our system are listed in italics in the said appendix.
7.6 Conclusions

We can make two general observations regarding the design of the implementation. Firstly, we note that the agents are primarily used to integrate and share the information available in the electronic environment of a user. Furthermore, FLC is used partly to keep messages that belong to the same conversation together in a logical sequence, and partly to organise limited semantic information about the communicating parties and make it retrievable for classification purposes. Secondly, two alternative scenarios are possible: symmetric and asymmetric. In the symmetric scenario both the sender and the receiver are using the implementation—in the asymmetric scenario only one of them is using it.

In the symmetric variant, both communicating parties have the full ability to keep track of the thread in a specific conversation, and thus the task involved. On the other hand, in the asymmetric variant, some more support is needed to make use of FLC, since incoming (outgoing) messages do not contain any FLC structures. A combination of using e-mail standards, semi-structured templates, text matching (and possibly recognition of proper nouns in full text), and an ML algorithm is required to keep a conversation together.

We note that the message-based organisation of information is kept intact in our approach. That is, information is stored and retrieved based on messages.

In the next two chapters we describe our system from two viewpoints: the developer’s and the user’s—in Chapter 8 we describe the implementation components of our system in detail, while in Chapter 9 we describe the functionality of our system from a user’s viewpoint by going through a working example.
Chapter 8
Implementation Details:
A Developer’s View of the Prototype

In this chapter we present implementation details of the prototype. After the introductory section we start describing how FLC is introduced into the e-mail system and subsequently to the messages. XML has a role in conjunction with this, which we explain. We continue with showing how a message is parsed and subsequently classified into a folder and (or) a thread. The function of RIPPER, the ML algorithm that we have chosen, is explained in this context. Thereafter we describe how the desktop sources in the electronic environment are harmonised. The JAVA-based agent infrastructure that we have chosen, called JATLITE, is described. We also show how the infrastructure is set up in the context of the e-mail client. We conclude the chapter putting all the components described in the chapter together.

8.1 Introduction
We used rather flexible programming tools for constructing a prototype. Prototyping was done mainly in TCL/Tk (Ousterhout 1994; Welch 1997) and PERL (Wall, Christiansen, & Schwartz 1996), with some programming in UNIX shell scripts and AWK.

As a platform, the EXMH e-mail client was selected. EXMH (Welch 2001) is used by other persons at our department, which opens up the possibility for a more extensive future evaluation of our system in a real environment. EXMH was extended using several of its mechanisms for customisation. We used mainly the function of so-called hook points pro-
vided in several key places in the EXMH implementation, which make it possible to add a personal library of TCL procedures.  

Why do we use EXMH? EXMH is built with the assumption that the user will want to customise it to some degree. The implementation is open and flexible—EXMH has been public software since 1992 and several programmers contribute code to it. Furthermore, EXMH is designed for high-volume e-mail users, with the aim to provide support for “efficient dealing with several hundred e-mail messages each day” (Welch 2001). EXMH has been integrated with PGP, GLIMPSE (Velez 2001), ISPELL, MIME, and many other software. New software, including filtering systems, can be added. These facts are all along our requirements for an e-mail system that we can extend with new functionality for implementing our prototype.

8.2 Structuring messages

A central question in this section is: “How do we introduce the message types, defined in FLC, into the e-mail client without increasing the cognitive load of the user?” As discussed in the previous chapter, we use message templates. Subsequently, the FLC information is put into a message by the use of MIME-based message attachments. We explain the details regarding the use of message templates and MIME attachments below. We also describe the role of XML in the context of structuring messages.

8.2.1 The message templates

We use message templates to enable the sender to structure and classify outgoing messages using FLC. By giving a message a preliminary structure and some initial contents, the probability of the message later being classified correctly becomes higher.  

By using a template when composing a new message the sender also states that she is starting a task.  

---

36. TCL stands for Tool Command Language and is really two things: a scripting language, and an interpreter for that language, designed to be easy to embed into an application. TCL and its associated graphical user interface tool kit, TK, were designed by Prof. John Ousterhout of the University of California, Berkeley. The TCL interpreter has been ported from UNIX to DOS, WINDOWS, OS/2, NT, and MACINTOSH environments. The Tk tool kit has been ported from the X WINDOW SYSTEM to WINDOWS and MACINTOSH.

37. The components file is used for creating the templates in EXMH (Peek 1995).

38. If the user is doing a reply and using a template then by default she is not starting a task since the pendingTask component in FLC will have the value “yes.” See, e.g., message 3 from person 2 in the scheduling-a-meeting scenario in Figure 5.6 on p. 101.
stored in the Tasks folder, which is the folder for pending tasks. The eight message types presented earlier (see Table 5.1 on p. 86) have been implemented. For example, the template for the APP message template (a request for a meeting appointment) contains placeholders for information about the meeting’s date, time, and place.

The templates are used to add the FLC structure to the message. In practice this information comes as an XML-encoded attachment—the user can choose if she wants the attachment to be included in the outgoing message, or just stored locally, along with the message in the Tasks folder. By using XML the attachment can be read as is, since it is plain ASCII—if any of the communicating parties, e.g., should happen to not have an XML reader available.

With regard to handling the message templates, we do text matching in a way similar to CAP (see Section 6.2.9 on p. 125). However, the keywords we are looking for are placed there by the message templates.

### 8.2.2 Including FLC in Messages

The FLC-based information is included in each message as an XML attachment. This XML-encoded representation follows FLC closely and consists of a series of individually meaningful and basically arbitrarily ordered declarations or statements.

For example, the FLC message 5.14 from p. 109 is shown in its XML format in Figure 8.1 on p. 160. It represents an e-mail message containing a note from the user to herself about the successful signing of a meeting minutes. The DTD corresponding to the document in Figure 8.1 is listed in Takkinen (2001). Since XML requires a root element, we have added the flcMsg (read as FLC message) for this purpose. Otherwise the XML representation closely follows the language FLC described in Chapter 5. Each XML component has an ID attribute for referencing purposes. The msgSt (message statement) element has sender, receiver, and attitude as attributes. The content of the message consists of a predSt (predicate statement) element containing the note predicate. The note contains an element for the person who has written the note, the time (including date), and an explanation in stringContent. The last part of the message consists of the context element. The attributes of context declare what the sending machine and task name are (sendingMachine and taskName, respectively), in addition to specifying that this is an ongoing task (the pending-Task attribute has value “yes”).

In this message, context has three parts. The first part lists the resources referenced in the message. These are classified into the two
groups actor and artefact. The only actor in this example is the person mentioned above, identified by the value of the ID attribute in actor. The artefact in the message is the message itself. The artefact can be a “document” or another FLC message. The alt attribute of artefact can be used to describe the content of the artefact—in the example it is the title of the minutes. The resources element has been added to the DTD so that the IDs of persons, messages, etc. used in the message can be double-checked for uniqueness. That is, the IDs listed as attributes in the elements (actor and artefact) in the resources section of the XML document represent the IDs that need to be unique in the document. This is accomplished by defining them as ID and #REQUIRED.

39. The document can be a file, an URL, among other things.

160
They are subsequently referenced by ID attributes declared as IDREF and #IMPLIED. See, e.g., the person element in Figure 8.1, which references the actor in resources (cf. Takkinen 2001). A validating parser, such as XEENA (“alphaWorks” 2000), verifies that every ID attribute referenced by IDREF is in the XML document. The third part of the context of the example XML message in the figure above consists of the timeSent element.

The structure of the predicates that we specified for FLC in Chapter 5 (absent, actionRequired, addItem, etc.) is also defined in the XML DTD (Takkinen 2001). In order to be able to handle the XML structure and the information contained in it, three approaches are used in the prototype:

- TCLXML (2001), a range of TCL-based (Ousterhout 1994) specifications and tools to process and manipulate XML structures
- SGREP (Structured GREP; Jaakkola 1998), a GREP-like information retrieval tool for retrieving structured information, is used for extracting not only XML information, but also information from the message itself
- XEENA (“alphaWorks” 2000), a validating XML editor, is used to display the XML information graphically.

TCLXML, originally authored by Steve Ball from Zveno, has a layered architecture similar to SAX (2001), “The Simple API for XML.” TCLXML includes two parsers: a TCL (Ousterhout 1994) interface to James Clark’s expat XML parser (Clark 2000) called TCEXPAT, as well as a parser written in TCL which has equivalent functionality to TCEXPAT, known as the native TCLXML parser. Furthermore, TCLXML includes a specification for a programming interface to manipulate XML (and HTML) documents based on the DOM called TCDOM.40 TCEXPAT allows a TCL script to parse an XML document using EXPAT, invoking TCL code as callbacks for various parts of the document such as start and end of element, processing instructions (PIs), declarations, etc. (see “Extensible Markup Language” 2001 and Bradley 2000 for more about XML).

SGREP is a tool for making queries to almost any kind of text files with some well-known structure. SGREP enables the searching and indexing of text, SGML, XML, and HTML files, and filtering text streams using structural criteria (Jaakkola 1998). For example, to extract all actor elements from within a resources element in file containing an XML-encoded

40. DOM—Document Object Model—is a W3C specification for application program interfaces for accessing the content of HTML and XML documents.
FLC structure (and print them each on a new line) we can do the following in SGREP (command broken into two lines because of space limits):

```
sgrep -o"%r\n" '"<actor" .. "">'
    in ('"<resources>" .. "</resources>"')
```

SGREP was selected since it is smaller and faster to use in a prototype than a full-fledged XML query language, such as XQL, the XML Query Language developed by the World Wide Web Consortium (Robie, Lapp & Schach 1998). Another alternative is XTRACT (2001), a GREP-like tool for XML documents, developed by Canon Research Centre Europe. XTRACT is a query language similar to XQL, but smaller and with some own extensions. Another way to query the XML structure is to use the XPath (XML Path) language, a recommendation from the World Wide Web Consortium ("W3C" 2001). It defines a syntax for locating parts of an XML document, e.g., attribute values. XPath is a string-based language of expressions used by other XML technologies, such as XSLT mentioned above, and the XML Pointer Language (XPOINTER) (see Cover 2001).

As mentioned above, XEEENA (XML Editing Environment, Naturally in Java) is an XML document editor. It is developed by IBM Research Laboratory in Haifa in Israel. XEEENA is a validating XML document editor, i.e., it checks whether a given XML document conforms to the accompanying DTD or not (cf. Bradley 2000). XEEENA allows the user to view and edit an XML document in a graphical tree mode. We use the editor as an interface to modify and store the message and task structures. An example is shown in Figure 8.2 on p. 163. The FLC message from Figure 8.1 on p. 160 is being edited using XEEENA. Here, the `predSt` part of the message is being edited. The attributes are shown in the window below to the left (only the `id` attribute is relevant). The next window to the left shows the available elements for `predSt` in bold. 41

The XML attachment of an e-mail message containing the FLC structure for the message can be displayed either by using a TCL command that calls TclXML, or by running XEEENA. The preferred action to take can be defined by the user in the Preferences in our system (see Figure 8.3 on p. 164). We have defined a DTD for the FLC used in the prototype (see Takkinen 2001). Our DTD is inspired by the DTD available from Moore (2001).

---

41. Some of the elements have associated icons, which can be defined either on a general level or specifically for each DTD.
With regard to the transportation of XML documents\textsuperscript{42} in e-mail messages (using MIME), the best approach for labelling these is to use a separate MIME media type for XML. Five media types for XML are standardised in RFC 3023 (Murata, St. Laurent & Kohn 2001): \texttt{text/xml}, \texttt{application/xml}, \texttt{text/xml-external-parsed-entity}, \texttt{application/xml-external-parsed-entity}, and \texttt{application/xml-dtd} (see Figure 8.2: The message from Figure 8.1 displayed using XEENA).

\textsuperscript{42} Unprocessed, source XML documents, readable by casual users.
for details about these types). RFC 3023 also standardises a convention for naming media types that are based on XML, but which are not identified by any of these five types. In these cases the suffix +xml should be used. By following this convention, an XML-based media type can be recognised. According to the same standard, the type text/xml is preferable to application/xml for unprocessed, readable XML documents. E-mail clients that do not have support for XML will treat text/xml as plain text/plain, i.e., displaying the XML document as it is.

In the prototype we therefore use text/xml to identify XML documents. Furthermore, we use the content type multipart/mixed to structure the e-mail message so that it can carry both the original message in plain text and the FLC information encoded in XML. An alternative would have been to use the content type multipart/alternative, because in a way we are really sending the same message in two different formats: one untagged, plain text, and the other XML-encoded FLC representation. However, to keep things simple and allow additional, e.g.,

43. For example, MathML is a markup language for mathematics documents and therefore the media type should be application/mathml+xml.

Figure 8.3: Setting preferences for viewing XML-encoded attachments in our system.
clarifying and inferenced, information to be put in the XML structure we elected to use the first approach.

8.3 Classifying messages into threads and folders

When the system is first installed, it uses the messages the user has currently stored in her folders to bootstrap its classifier. The classifier is based on RIPPER (Cohen 1995; Cohen 1996). After its initial training, the system continually monitors changes to the user’s e-mail database and regularly updates its classifier. Specifically, the system immediately learns about any new folders and predicts that future messages about, e.g., racquetball should go in the “Racquetball” folder.

However, before we describe the details of RIPPER and how its rule-generating capability is used in our system, we explain the details of the enhanced threading mechanism.

8.3.1 Locating a matching thread

The algorithm for locating a suitable thread for a message (an annotation, a text file, etc.) does matching on different parts of the message ordered as follows:

- the task name given in the FLC structure, if available, which is matched against any task names in the other threads
- the message ID in the In-reply-to: header of the message being inserted, which is matched against the IDs in the Message-id: headers of the messages already in the Tasks folder
- the subject of the message, available in the Subject: header, which is matched against the other Subject: headers
- the text in the body of the message, excluding attachments encoded in base64 but including the To:, From: and Cc: headers, which is matched against the text in the body of the other messages, and ranked using TF-IDF weighting (term frequency—inverse document frequency).

In the above cases of matching against task name and subject, respectively, the latest message matched is the one suggested to the user (the messages are sorted according to date before doing the matching). In the last case of matching on the body of the message the highest-ranking message according to the calculated TF-IDF ranking is selected.

---

44. The message IDs are at least in theory globally unique, all according to the Internet e-mail protocol standard (Resnick 2001).
A message must have a \texttt{Message-id:} header to be included in the threading algorithm. This is also what is required for an e-mail message by RFC 2822 (see Chapter 3). If it is not available, \textit{e.g.}, if a draft message or an annotation is inserted into the thread, then a unique \texttt{Message-id:} header is created based on the current date and time, including seconds, and the local machine’s name.

The task name is initially set to the contents of the \texttt{Subject:} header, but can be changed by the user. Common e-mail “netiquette” advocates that a user should always have a relevant subject on e-mail messages. By initiating the task name to the contents of the \texttt{Subject:} line our system emphasises the importance of using relevant subjects since it will show up in the list of tasks. In conclusion, the task name represents \textit{a way of naming threads}.

The contents of the \texttt{Subject:} header are normalised before any matching is done. This means that the text is converted to lower case, decoded from quoted-printable MIME encoding if applicable, and all \texttt{Re:} tokens removed—the so-called reply markers—in addition to punctuation and some special characters.\footnote{We also include the \texttt{Sv:} token, the Swedish variant of the \texttt{Re:} token. Punctuation and special characters include periods, commas, colons, semicolons, single quotes, question marks, exclamation marks, equal signs, underscores, and the Swedish å, ä, and ö.} Also, tab characters and space characters are both replaced with one single space character instance.

Matching text in the body of messages based on TF-IDF weights is used as a last resort, when all the other matching techniques described above fail.

In this case each message in the Tasks folder is converted into its TF-IDF representation, a message surrogate. We rank the messages using a variant of TF-IDF and the cosine similarity formula described by Salton and colleagues (1988; 1996). The message representation consists of a vector of numeric weights \(<w_{i1}, w_{i2}, \ldots, w_{ik}, \ldots, w_{it}>\) where \(w_{ik}\) is as follows

\[
w_{ik} = \frac{f_{ik}}{\sqrt{\sum_{j=1}^{t} f_{ij}^2}}
\]
The variable $f_k$ is the number of times word $k$ appears in message $i$. The message which is moved to the Tasks folder is treated as a query message, which also is represented as a vector: $<q_1, q_2, ..., q_k, ..., q_t>$ where

$$q_k = \frac{f_k \times \log(N/n_k)}{\sqrt{\sum_{j=1}^{t} (f_j \times \log(N/n_j))^2}}$$

Here, $f_k$ is the number of times the word $k$ occurs in the query message, $N$ is the number of messages in the Tasks folder, and $n_k$ is the number of messages containing word $k$.

All words longer than three characters in the body of each message are included in the algorithm, counted and listed in frequency order (TF). Then the IDF for each word is calculated, i.e., the value $\log(N/n_j)$, where $N$ is the total number of messages in the folder and $n_j$ is the number of messages where the word (term) occurs. The message in the Tasks folder with the highest score relative the query message is the match. The score of each message $i$ is calculated as follows:

$$\sum_{j=1}^{t} q_j w_{ij}$$

When a matching message has been located, the task name of the thread that the message belongs to is suggested to the user, along with a list of the names of the available, pending tasks in the system. The user can either accept or reject this suggestion. If she accepts the suggestion a `References:` header containing the ID of the matching message is added to the message before it is moved to the Tasks folder and inserted into the thread.

The above procedure applies if the match was based on either the task name, the subject, or the body of the message. If the message was matched on a message ID occurring in an `In-reply-to:` header then the message already has a reference and is just moved to the folder where the corresponding thread resides.

The task name in the FLC structure of the moved message is updated to the one available in the matching message. If an FLC structure is not avail-
able in the moved message then it is created and added to the message, based on the message type Other (OTH) with the force unknown (cf. Figure 5.4 on p. 94).

8.3.2 Generating automatic filtering rules

We use automatically created filtering rules to sort incoming messages into folders. The rules are in the form of IF–THEN rules that look for key-words in the headers of the messages. The learning procedure for the automatic filtering rules is summarised in Figure 8.4. Central in the procedure is the machine learning (ML) algorithm RIPPER (Repeated Incremental Pruning to Produce Error Reduction). In our system, RIPPER needs as input a file messages.data with the message surrogates and a file mes-

1. Create/Truncate the file messages.data
2. For each folder listed in messages.names
   • For each message in folder for which
     age of message ≤ two months
     and folder not listed in messages.skipme
     – Extract keywords from the headers of message
     – Append the keywords, followed by the folder name,
       as a row to messages.data
3. Run RIPPER, with messages.data and
   messages.names as the input. RIPPER generates a
decision tree and converts each path into a rule and stores
the rules in message.hyp
4. Initialise .procmailrc for new PROCMAIL rules
5. For each RIPPER rule in messages.hyp
   • Parse and extract folder name, attribute name, and
     attribute value
   • Rewrite the rule in the PROCMAIL format and store
     it in .procmailrc, including an action that adds an
     X-rule: header with a rule description to each
     message sorted by the rule
   • If the empty rule is given as the default rule by
     RIPPER, replace it with the EXMH inbox in PROCMAIL.

Figure 8.4: The learning procedure for the filtering rules in our system.
**IMPLEMENTATION DETAILS: A DEVELOPER’S VIEW OF THE Prototype**

*sages.nam**ewith the target folders. The rules are converted into a format that can be used by PROCMAIL (2001). PROCMAIL is a well-known and versatile package for filtering, which in turn EXMH can make use of.

RIPPER represents a successful approach of employing a machine learning algorithm for learning “keyword-spotting rules,” e.g., for use in e-mail. It was developed by William Cohen (1995; 1996) based on the repeated application of IREP (Incremental Reduced Error Pruning) algorithm (Fürnkranz & Widmer 1994). RIPPER learns filtering rules that use keywords for sorting messages into categories. It is a decision tree-based machine learning algorithm written in ANSI C for UNIX systems. Consider the basic case of classifying an example as either positive or negative. The RIPPER algorithm is shown in Figure 8.5. The figure shows a high-level description of the algorithm, with the original steps of the IREP algorithm and the additions made by RIPPER in bold face. RIPPER adds

<table>
<thead>
<tr>
<th>LOOP n TIMES:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Start with the empty rule (TRUE ⇒ positive)</td>
</tr>
<tr>
<td>2. LOOP UNTIL the stopping condition is reached.</td>
</tr>
<tr>
<td>• Partition the training set into a growing set and a pruning set</td>
</tr>
<tr>
<td>• Grow a rule by greedily adding a clause to the left-hand side guided by the grow heuristic</td>
</tr>
<tr>
<td>• Prune a rule by greedily deleting sequences of final clauses guided by the prune heuristic.</td>
</tr>
<tr>
<td>• Remove examples covered by the rule from the training set.</td>
</tr>
<tr>
<td>3. END LOOP.</td>
</tr>
<tr>
<td>4. Perform rule optimisation on the entire rule set.</td>
</tr>
<tr>
<td>END LOOP.</td>
</tr>
</tbody>
</table>

**Figure 8.5:** The RIPPER algorithm step-by-step (bold face indicates additions to the original IREP algorithm).

several rule optimisation steps, as well as an option to repeat the entire process to improve the rule set. The growing and pruning of IREP, which is used in RIPPER, means that the rules are made more restrictive (fitting the data more closely) and less restrictive, respectively. The pruning is done to avoid overfitting. Overfitting occurs when the algorithm fits the training data very well, but it performs poorly on unseen data. The training
set of the algorithm is split into one growing set and one pruning set. The positive rules, used to predict the positive class, are grown one at a time by starting with the empty rule and adding clauses to the left-hand side using the grow heuristic. The growing of a single rule is stopped when it covers no negative examples from the growing set. Each rule is pruned immediately after it has been grown by deleting clauses that cover too many negative clauses in the pruning set (a prune heuristic is used). When a new rule has been grown and pruned, all of the examples that the rule covers are removed from both the growing and pruning set. The rule growing process is repeated until all examples in the training set are covered or until some stopping condition is met.

RIPPER handles multiple classes $C_1$ to $C_k$ by ordering them from the least to the most prevalent and then treating each class in order as a distinct two-class problem. That is, RIPPER first learns rules to distinguish the least prevalent class $C_1$ from classes $C_2$ to $C_k$. After that, all the examples that are covered by the rule set are removed, and the process is repeated. The last class is used as the default class. This ordering of classes made by RIPPER means that rules will always be learned to cover the positive class first, and the negative class will be left as the default.

### 8.3.3 The Message Surrogates

As noted above, RIPPER needs two kinds of files with data: one file with the message surrogates (also called the document vectors), and one file with the targets. The message surrogates are the messages as represented by the keywords (tokens) found when parsing. In our system, these message representations are stored in the file `messages.data`. The targets are the names of the user’s folders, and they are stored in `messages.names`. The keywords are extracted using TCLEX (Bonnet 1999), a TCL-based lexical analyser (lexer) similar to UNIX and GNU LEX and FLEX, all tools for generating programs that perform pattern-matching on text. Since TCLEX is written in TCL, it is easy to include and use in the EXMH code that we customise for our system.

One question when constructing the message surrogates is “Which vocabulary of message attributes should we consider with regard to RIP-

---

46. RIPPER uses the information gain function (Quinlan 1990) as the grow heuristic.
47. The Minimum Description Length (MDL) principle is employed as the stopping condition (Quinlan & Rivest 1989).
IMPLEMENTATION DETAILS: A DEVELOPER’S VIEW OF THE PROTOTYPE

171

PER?” Potentially, we have all the words in a message available. However, we have performance to consider, as well as feasibility.

Since e-mail messages are mostly short messages, the Subject: line of a message tends to tell a lot about the contents of the message (Takkinen 1997). According to an investigation of readers’ selection of net news articles, performed by Ann Lantz in the INTFILTER project of Stockholm University (Kilander, Fähræus, & Palme 1997), the main criteria for deciding which article to read are the Subject: header and the genre or type of the article. The main criteria for skipping an article are the Subject: header, the length of the article and the amount of quoted material in the article. According to Fleming and Kilgour (1994), the most referred headers for reading an e-mail message are the From: header, the Subject: header, and the To: header (and to some extent also the Date: and Cc: headers).

A typical e-mail message from a mailing list (cf. Section 5.3.1 on p. 85) is shown in Figure 8.6 on p. 172. Note the large portion of headers compared to the amount of text in the message body. The corresponding message surrogate used in our system is shown in Figure 8.7 on p. 173. The format is understood by RIPPER. In the message surrogate (see Figure 8.7) the header attribute (from, to, etc.) and its value are separated from each other by the underscore character (“_”). Furthermore, the attribute–value pairs are put within single quotes, and separated by a space character. The last item within single quotes in the message surrogate is the target folder for the message. Each message surrogate ends with a dot.

To keep the number of keywords for each message representation low, we decided to limit the extraction of keywords to certain headers: To:, From:, Cc:, Subject:, Sender:, Date:, and all headers matching the pattern X-*: (such as X-url:). From each header we extract “alphastings”—strings of alphabetics, i.e., no numbers—consisting of four or more letters, with the exception of the Date: header, where we extract the day-of-month number, apart from the (three-letter) day name, and the (three-letter) month name. We also limit the number of messages parsed from each folder to those less than two months old. One reason for this restriction is the requirement for acceptable performance of the ML algo-

48. The addressee in the Sender: header is in most cases the same as in the From: header. However, in some cases they can be different. For example when a secretary has written a message on behalf of another person—in that case the secretary’s address will be in the Sender: and the other person’s in the From:.
CHAPTER 8

From exmh-workers-request@redhat.com Thu Feb 24 02:09:51 2000
Received: from lists.redhat.com ([lists.redhat.com (199.183.24.247)])
    by portofix.ida.liu.se (8.9.3/8.9.3) with SMTP id CAA22030
    for <juhta@ida.liu.se>; Thu, 24 Feb 2000 02:09:49 +0100 (MET)
Received: (qmail 401 invoked by uid 501); 24 Feb 2000 01:09:47 -0000
Resent-Date: 24 Feb 2000 01:09:47 -0000
Resent-Cc: recipient list not shown:
MBOX-Line: From exmh-workers-request@redhat.com Wed Feb 23 20:09:47 2000
X-Image-URL: http://www.ipass.net/~klassa/face.gif
X-Alternate-Address: klassa@ipass.net
X-URL: http://www.ipass.net/~klassa
X-Mailer: exmh version 2.1.1 10/15/1999
From: klassa@mail.com (John Klassa)
Reply-To: klassa@mail.com (John Klassa)
To: Anders Lund <Anders.Lund@uninett.no>
cc: exmh-workers@redhat.com
Subject: Re: QP fix for cite
In-reply-to: Your message of "Sat, 19 Feb 2000 17:06:19 +0100."
    <200002191606.RAA20756@tyholt.uninett.no>
Date: Wed, 23 Feb 2000 20:08:29 -0500
Message-ID: <1951.951354509@cisco.com>
Sender: klassa@cisco.com
Resent-Sender: exmh-workers-request@redhat.com
Resent-Message-ID: <"DHRxa1.0.16.QJ8ju"@lists.redhat.com>
X-Mailer: exmh version 2.1.1 10/15/1999
X-URL: http://www.ipass.net/~klassa
X-Alternate-Address: klassa@ipass.net
X-Loop: exmh-workers@redhat.com
X-List: exmh-workers@redhat.com archive/latest/128
X-Precedence: list
X-Mailer: exmh version 2.1.1 10/15/1999
X-URL: http://www.redhat.com
List-Id: <http://www.xray.mpe.mpg.de/mailing-lists/exmh/>
List-Maintainer: <mailto:exmh-workers-request@redhat.com>
List-Post: <mailto:exmh-workers-request@redhat.com>
List-Unsubscribe: <mailto:exmh-workers-request@redhat.com>

Anders,
I'm too swamped to take a good look at what you've done... Rather than
hold up progress, and in the name of all that is "Open Source" :-), you
have my permission to do whatever you think best with it and submit it
for a commit into the CVS tree. My apologies... Work has just been
crazy lately, and will continue to be for at least another week or two. :-/

[ Exmh-Workers: If you're a regular cite.pl user, please have a look at
what Anders has done and give him some feedback as you see fit. Thanks! ]

Swampedly yours,
John
John Klassa / klassa@mail.com

Figure 8.6: A typical mailing list e-mail message.

172
IMPLEMENTATION DETAILS: A DEVELOPER'S VIEW OF THE PROTOTYPE

Algorithm (RIPEER): the more messages to parse, the more time is required and the larger the set of values to consider. The two-month limit seems to give us a manageable number of messages to work with. Mitchell and colleagues (1994) use a similar restriction in their CAP (see Section 6.2.9 on p. 125). The time interval needed to acquire sufficient data for reliable learning must be shorter than the time interval over which learned regularities remain stable. In CAP's case the first time interval (called the "learning time constant") was two months and the second time interval four months (called the "task time constant") (Mitchell et al. 1994).

8.3.4 USING RIPEER FROM EXMH

RIPEER is, in the prototype, typically run in the background at regular intervals, for example every hour. In our prototype this background processing can be set up as shown in Figure 8.8 on p. 174. The three lower rows in the figure show the additions to the Preferences section of EXMH where the execution of RIPEER can be controlled: the execution can be either off or on, the period (in minutes) can be set, and the corresponding script for running RIPEER can also be defined.

When we generate the filtering rules we take into consideration messages that are up to two months old (see Section 8.3.3 on p. 170). The conversion of the RIPEER rules into PROCMAIL's rule format is straightforward. One additional feature PROCMAIL offers is the option to modify a classified message: we can add an extra X-rule: header to the message. This header makes information about the rule used for the matching and filing of the message available to the user (see Figure 8.9).

By examining the X-rule: header the user can check why, for example, a message has been filed into the wrong folder. In the example in the

Figure 8.7: A message surrogate in our system, generated from the message in Figure 8.6.
figure the message shown has been filed to the current folder based on the fact that the From: header contained the keyword developer. When a message has been filed into the wrong folder and the user refiles it, the two data files for RIPPER (messages.data and messages.names) are accordingly updated with the keywords and the new target folder, so that

Figure 8.8: Setting up RIPPER to execute in the background in our system.

Figure 8.9: Displaying the triggered filtering rule as an X-rule: header in our system.
any similar messages can have a higher probability of being correctly classified the next time.

If the target folder of the refiled message does not exist in the messages.names file, it is included there. The exception to this is if the target folder is listed in a file named messages.skipme. This file contains names of folders which contain messages that for different reasons should never be included in the filtering algorithm.

8.4 Harmonising desktop sources

We have implemented a subset of the multi-agent architecture described in the previous chapter. We have based the implementation on JATLITE ("JATLITE" 2000; Heecheol, Petrie, & Cutkosky 2000), a package of JAVA classes and programs developed at Stanford University. JATLITE (Java Agent Template, Lite) allows a developer to create new software agents that communicate robustly over the Internet. The agents can be either new software created directly for JATLITE or legacy software that uses wrappers to send and receive agent messages (cf. Figure 7.3 on p. 149). JATLITE uses a layered architecture since that is the best to use for complex software of this kind ("JATLITE" 2000; Tanenbaum 1996). Agents register with an Agent Message Router (AMR) using a name and password. The AMR facilitates for the agents to connect/disconnect from the Internet, send and receive messages between each other, transfer files, and invoke other programs or actions on the various computers where the agents are running.

JATLITE gives us the infrastructure for integrating and sharing the information stored in the electronic environment of the user. Three simple agents for collecting and harmonising the information from three different sources have been implemented: one for indexing and managing calendar events, one for monitoring outgoing messages, and one for managing the address database in EXMH (the address book). The indexes include both a direct index file and an inverted index file. For example, the direct index file for outgoing messages is indexed on the message numbers, while the inverted index file is indexed on the keywords found.

A collection of one-shot PERL (Wall, Christiansen, & Schwartz 1996) and shell scripts are also included in the agent part of the system, in addition to the agents themselves, which are implemented in JAVA. The prototype for the agents has been implemented on a UNIX workstation running SOLARISOS.
The agents can be defined and managed in ExMH through the Preferences window (see Figure 8.10). In the lower left of the figure three agents are being set up: one called the **Calendar** agent, one **OutMess** agent, and one called the **AddrBook** agent. In addition to these agents an **AgentRouter** is also specified (top right window in the figure). This is the central component and obligatory one in the infrastructure because it routes the agent messages to and from the agents. The **Calendar**, **OutMess**, and **AddrBook**
agents index and manage the information in the calendar program, the outgoing messages folder, and the address book of the user, respectively (cf. Section 7.4.3 on p. 151).

Currently we do not make use of the agent infrastructure in the prototype, but we have tested the communication between the agents and their indexing of the information sources that they are responsible for. The infrastructure is an interesting subject for future work, in that it provides for the electronic sources to be available over the Internet, for example, to mobile users. Also, the classification of messages can be enhanced by combining clues available from the different—possibly distributed—electronic sources (cf. Marx & Schmandt 1996): we can include information from them when generating rules using RIPPER.

8.5 Putting it all together

We have presented the design considerations in the previous chapter, and in the current chapter described in detail the parts included in the implementation. We now conclude by describing how it all comes together in a prototype.

Regularly and automatically, messages in a number of folders are parsed for keywords—mainly words occurring in headers of the messages—to create message surrogates that can be used in the filtering rule generation part. In other words, the classification rules are updated by first creating the message surrogates and then using the information in the surrogates to update the rules database. The filtering rules are used for sorting the incoming messages into the appropriate folders. The incoming messages are also parsed and included in the message surrogate database. An index of these surrogates is created, for easier retrieval by both the filtering system and the agents. In addition, information in the outgoing messages, the data in the calendar, and the address book are also parsed and stored in a similar format as the message surrogates described above.

Message types and templates are used by the system to naturally interact with the user and implicitly (from a user’s viewpoint) input FLC information into the messages. The FLC information comes in the form of data in the headers of the messages created with a template, and the information input by the user in the message (and in placeholders for template-specific information, if any). The FLC information is encoded using XML and attached to the message. As a side-note, the agents in the system also make use of XML, to structure the information in the sources and make it...
easier to manage and retrieve. To summarise, the relation between FLC and XML is as follows:

- FLC takes care of the semantics, i.e., the relations between subject, sender, recipient, what the communicating parties are trying to achieve, etc.
- XML handles the syntax, i.e., marks up where the strings for the sender, recipient, subject, date, etc. are in the messages.

Finally, when formatting a message for presentation to the user, features (header fields) that are more important for the current type of the message are reinforced (or suppressed, if the features are less important). Furthermore, messages in threads representing tasks are indented, starting from the root message which initiated the task.

This ends the developer’s view of the prototype. In the next chapter we examine a user’s view of the prototype.
Chapter 9
A Working Example:
A User’s View of the Prototype

In this chapter we walk through a working example of how the scheduling-a-meeting scenario looks through an implementation of the conceptual model. After the walk-through, we separately describe two aspects that are not directly demonstrated in the scenario: how to observe (peruse) pending tasks in the prototype, and how to stop a task and archive it for later reference.

9.1 The scheduling-a-meeting scenario
The communication and its parties, including the message types used in the working example, are shown in Figure 9.1 on p. 180. It is only slightly different from Figure 5.6 on p. 101: message 4 is OTH instead of STM—the reason for this is explained in the text that follows. Please refer to the figure below when following the working example.

9.1.1 Starting a task
We assume that the user has the habit of storing her pending tasks in a separate folder, named Tasks, and that this folder is empty from the beginning. The initiator starts the meeting scheduling task by composing a new message, i.e., selecting the APP message template on the Comp button menu (see Figure 9.2 on p. 181). The user (the initiator) inputs the e-mail

49. The user could also have been using the inbox for the pending tasks, or one folder per task.
The addresses of the three other persons involved in the task in the To: header.\footnote{Because they are already a part of EXMH, the addresses can be looked up in the address database of EXMH by simply inputting a partial name or address and then pressing Control-Tab. If the name or address exists in the database it is either expanded, or a list of possible alternatives is shown if multiple matches exist.}

The message template for the APP message reminds the user about the information that is required for a request for a meeting. There are placeholders in the body of the message template for the date, time and place for the meeting.

After filling in the meeting information in the message and writing any additional information, the initiator clicks on the Send button. She is now asked if she wants the receivers in the To: (and any Cc:) headers to also receive the XML-encoded task (FLC) information that has been added to the message by our system (see Figure 9.3 on p. 182). She clicks on No, which means that the FLC information will only be stored together with the local copy of the sent message.

The message is now sent, without the FLC information as requested (see message 1a, 1b, and 1c in Figure 9.1). A copy of the message, including the FLC information, is moved to the Tasks folder. When the message is filed in the Tasks folder, the user is asked to provide a name for the task.

\footnote{For privacy reasons, some parts of the messages shown in the figures in the chapter have been masked.}
A Working Example: A User's View of the Prototype

The default task name when starting a task is the contents of the **Subject:** header. Because there are no other pending tasks in the Tasks folder (this was the introductory assumption), this is the only task name suggested to the user by our system.

The message in the Tasks folder now contains an XML-encoded attachment with the FLC information. When viewing the Tasks folder and perusing the message, our system/EXMH displays the message to the user as shown in Figure 9.5 on p. 183. The attachment has the text “View XML contents.” By clicking on the text the attachment is parsed and the information in it is presented. The action for our system to take when the user clicks on the attachment depends on the settings given in the XML section of the

![Figure 9.2: Selecting a message template from the Comp button menu in our system.](image)

52. This could perhaps lead to more meaningful **Subject:** headers.

181
Figure 9.3: Choosing in our system whether or not to distribute FLC information.

Figure 9.4: Naming a task in our system.
preferences (see Figure 8.3 on p. 164). Here, the setting is to pass the XML code to a user-defined Tcl command, defined by the \texttt{tclcmd} variable.

Let us now assume that the messages (the MDNs) from persons 2 and 3 have arrived (messages 2a and 2b in Figure 9.1 on p. 180). These two persons are also using the prototype (our system).

The messages are parsed and automatically sorted by \texttt{PROCMAIL} into folders according to the rules created by \texttt{RIPPER}. The user can see which rule was triggered by reading the \texttt{X-rule: header} (cf. Figure 8.9 on p. 174). When the messages are filed in the Tasks folder the user (in this case the initiator) will be requested to confirm that the messages belong to the “computer networks meeting” task. This is where the Threads Browser window is shown to the user (see Figure 9.6 on p. 184).

The default task name in this case is found in the FLC information attached to the message, \textit{i.e.}, “computer networks meeting.” This is shown at the top of the Threads Browser window. By clicking OK the user confirms that the message should be moved to the Tasks folder and that it

\textbf{Figure 9.5:} FLC as an XML attachment in a message in our system.

---

53. MDNs are a part of EXMH. In the preferences it can be controlled if and how one wants to send MDNs (receipts).
should belong to the task selected. The same Threads Browser window is shown if the user explicitly refiles a message from another folder into the Tasks folder.

The suggestion made by our system in the Threads Browser window is based on one of four criteria, and in the following order of precedence: the task name in the message’s FLC structure (if present), message IDs referenced in the headers, the Subject: header contents, or matching text in the body (cf. Section 8.3.1 on p. 165). An example of matching based on the referenced message IDs in the scenario is the reply sent by person 1 to person 0 (message 7 in Figure 9.1 on p. 180). Person 1 uses the Reply function in her e-mail client, and her e-mail client adds a reference to the old message.54

Let us stop a moment and assume that there are several tasks to choose from in the Tasks folder. For example, in Figure 9.7 on p. 185 there are two tasks pending in the Tasks folder. The user can select a task from the list, or click OK to confirm the suggestion made by our system (which was

---

54. Some e-mail clients do not include this reference. In these cases the matching on Subject: header contents is needed. Matching on text in the body is used as a last resort.

Figure 9.6: Filing the MDNs in the Tasks folder.
found using the four criteria listed above). The message is then moved to the Tasks folder and inserted into the selected task (thread). The FLC information is also updated (references to other messages, documents, etc.). The user can also abort the move by clicking Dismiss. Clicking on New Task will start a new task, as shown in Figure 9.8 on p. 186.

Continuing with the scenario, when person 2 has received the APP message from person 0 she almost immediately starts to compose a reply (message 3 in Figure 9.1 on p. 180). Person 2 selects the template named Statement for the STM message type on the Reply button menu—this menu is identical to the Comp button menu shown in Figure 9.2 on p. 181—and clicks Send. Now, both the reply message and the replied-to message—if not filed there already—are moved to person 2’s Tasks folder. Eventually both of the messages are inserted into an appropriate task (thread) in the Tasks folder, following the same dialogue as shown in

55. Person 2 is also using a Tasks folder for compiling pending tasks.
Figure 9.6 above. 56 Message 4 in Figure 9.1 on p. 180 is the reminder, which is triggered according to the instructions given by the initiator to the calendar program (see Section 5.5.1 on p. 101). The reminder is a message sent by the calendar program, classified as an OTH message type by our system (see Figure 9.9 on p. 187). When seeing the reminder the user is (implicitly) reminded to check the status of the task. Note that the message type differs from the one used in Chapter 5. This is because the user cannot (currently) set the message type in the calendar program. Instead these are classified as Other when they are received in our system, which is the default for non-FLC messages.

56. Regarding where the copy of a reply to a message should go, EXMH can easily be configured to automatically store a copy of the reply message in the same folder as the message being replied to. This can be accomplished by using the Fcc: header, which puts a copy of outgoing e-mail into a specified folder. The current folder is represented by the @ symbol. Consequently, a copy is stored in the current folder by adding the line Fcc: @ to the replcomps file (the template file for replies).
After seeing the reminder, person 0 notices that person 1 has not responded in any way to the meeting request. However, after a telephone call person 0 gets an acknowledgement from person 1. In addition to this, person 1 decides to write a response to the original APP message. Person 1 is assumed to be using a proprietary e-mail client with no support for FLC or XML. When the answer finally arrives at the initiator it can also be filed into the Tasks folder, into the right thread. Since this message is created by a non-FLC system, it is classified as a message of type OTH (Other) (the default for “unknown” messages). FLC information, encoded in XML, is added to the message and subsequently updated with information from the task.

Lastly, after the meeting minutes have been signed (messages 8 and 9 in Figure 9.1 on p. 180), person 3 makes a note about this in her system. She does this by sending herself a message containing an explanatory text, and the electronic version of the meeting minutes attached. Person 3 creates the message by selecting the original APP message and replying to it, and changing the To: header contents to her own e-mail address.

Regarding replies there is a task-related enhancement available in our system. When a message in a pending task is being read, it is possible to

57. More specifically, attitude unknown and message type OTH are the default if no X-Message-Type: header is present in the message (see, e.g., the headers shown in Figure 9.3 on p. 182).
employ the Reply to Task Group function. This is an extension of the Reply function, which inserts the addresses of all the participants in the task into the To: header. To reply to the whole task group, the user can select (mark) a message that she wants to reply to and then click the Reply menu button. There she can select the Reply to Task Group function and an appropriate message template for the reply message. Figure 9.10 shows what it looks like. As is customary in replies, the To: header is filled in by the system, but this time containing the e-mail addresses of the people involved in the task. The addresses are fetched from the messages belonging to the task thread.

Figure 9.10: The Reply to Task Group function in our system.

Congruent with the Reply button menu, there is a Compose to Task Group function on the Comp button menu available when composing new messages, as shown in Figure 9.2 on p. 181.

9.1.2 Observing a Task

Messages belonging to pending tasks are stored in the Tasks folder. Typically, the messages in a task have the same Subject: content (with an added Re: token in front). However, to really see which messages belong to which tasks, the user needs to see the threads in the folder. The following four thread view modes are supported in our system:

1. **All Tasks Threaded:** The user can select Thread from the More but-
ton menu in the middle of the EXMH window, and thereafter Full Threads from the submenu (see Figure 9.11). The messages in the Tasks folder are then rearranged and displayed as threads, formatted as shown in Figure 9.12 on p. 190. The “<-->” token marks the next message in the thread—if the token is indented then the message in question is related to the previous message.\(^58\)

2. **One Task Threaded:** To only see the messages belonging to one of the tasks, the user can mark one of the messages in the task in question and select Display Related Messages from the Search button menu, which is just to the left of the More button menu in the middle of the three-paneled EXMH window. The result is shown in Figure 9.13 on p. 191.

---

\(^{58}\) This way of showing threads is the default in EXMH (Welch 2001). The relations between the messages in the threads are found using the threading algorithm described in Section 8.3.1 on p. 165.
3. **Only Thread Roots:** The user can elect to see only the root messages of the threads. This is done by marking a message and selecting Threads and Only Root from the More button menu. The result is shown in Figure 9.14 on p. 191. With this function the user can get an overview of how many different tasks are pending.

4. **No Tasks Threaded:** By selecting Rescan Folder from the More button menu (see Figure 9.11), the display reverts back to the unthreaded view. This is the default when initially viewing a folder.

To get a summary and an overview of the tasks active in the system, the user can also select Task Info on the Tasks button menu. A separate window with task information is shown. The task information is compiled from the information found in the messages stored in the Tasks folder and includes a listing of the following items:
A Working Example: A User’s View of the Prototype

- the task name
- the date of the first message (the start date)
- the date of the last message, which is the completion date if the task has been finished
- the persons involved in the task, which is a list of e-mail addresses and aliases, along with the number of messages that they have sent each
- the person responsible for the task (the initiator in the scenario), which defaults to the sender of the first message in the thread
- a list of document (attachment) names and clickable links to the physical copies found in the messages and (or) the web

Figure 9.13: The Display Related Messages function, for displaying one thread at a time in our system/EXMH.

Figure 9.14: The result of the Only Root command in our system/EXMH.
• the status of the task, which can be “pending” or “completed”
• any associated programmed reminders and their timeout values.

Usually the person responsible for a task is the initiator of the task, but it could also be another person. This could be the case when a task has been delegated.59

9.1.3 Ending a Task and Saving Task Information

When all involved persons have confirmed the meeting, the scheduling-a-meeting task can be stopped in the following way:

1. The user marks one of the messages in the Tasks folder, representing the task.
2. She then selects a folder in which to archive the task with a click on the right mouse button. The colour of the messages previously marked is changed when the user clicks with the right mouse on a destination folder.60
3. The user thereafter issues the Stop Task command by selecting it from the Tasks button menu. The whole thread that the marked message belongs to will now be moved to the selected folder.

This sequence of mouse-clicks and command selection is in line with the other commands in EXMH, such as the Link command (see Peek 1995), which is used for creating a linked copy of a message. In step two above, if the user does not have a suitable folder available, she must create the folder first, which is done the usual way in EXMH by clicking the New button in the middle panel (cf. Figure 9.2 on p. 181). An alternative way for stopping a task is the following:

1. The user marks one of the messages in the Tasks folder, representing the task.
2. She then selects Display Related Messages from the Search button menu (see Figure 9.13 on p. 191). The messages belonging to the thread, representing the task are shown in the message summary window of EXMH.
3. The user then marks all of the messages in the shown thread by dragging with the mouse while keeping the left mouse button down. She can then point and click with the right mouse button on a folder where

59. See the making-conference-arrangements scenario in Appendix A.
60. This is part of EXMH’s behaviour and not specific to our system.
the marked messages should be archived.
4. By clicking Commit the marked messages are then moved to the se-
lected folder.

In other words, the first alternative above is the explicit way of stopping a
task, while the second alternative is the implicit approach.

9.1.4 The Task Database
The Task Database (TDB), which contains information about previously
performed tasks, has not been implemented in the prototype. Here is a sug-
gestion to how it would work.

When the messages associated with a task have been stored in a folder,
the TDB can be viewed. In the TDB all previously saved tasks can be
edited and then saved again. Since the FLC information is encoded in
XML, it is possible to implement different presentation formats. The user
selects the TDB from the Tasks button menu. A separate window appears
containing a list of names of all the tasks the system has stored informa-
tion about. The user can now click on a task and browse the FLC structure
of it, message by message. The task structure contains the task information
from the Task Info window, and in addition, FLC-specific information,
such as the message types involved, the attitudes, and predicates, among
other things—including references and links between messages. The user
can select any of the following actions in the TDB:

- name and save the (recently ended) task as it is, represented by the
currently marked message
- edit and name the structure of the marked task, and save it
- look for similar tasks and possibly merge them
- dismiss the task database window without saving the current task.

9.2 Summary and conclusions
To summarise, a task is initialised at the user’s request in either of two ways:

- The user replies to an existing (and already parsed and indexed) mes-
sage. The reply and the replied-to message are both placed in the
Tasks folder.
- The user composes a new message using a semi-structured template rep-
resenting the message type (see Table 5.1 on p. 86). The template in turn
defines the FLC information (see Chapter 5) to be put in the message.
The new message is copied into a thread in the Tasks folder. The message is also copied into the folder for outgoing messages, which is one of the sources included in the indexing of the sources in the electronic environment (see Chapter 8). In this way the conversation is made visible: the outgoing message is automatically included in the (growing) thread and not just stored—or even worse, discarded!—in the outbox. In other words, the notion of thread is very visible; it is a component in our system that the user can move and refile in much the same way as a single message.

Because the user may employ synchronous communication, e.g., the telephone or ICQ (“ICQ” 2001), in combination with e-mail to perform parts of a task—and even end a task—there has to be some way of explicitly marking a task (a thread) in e-mail as completed. The user explicitly tells the system when a task is finished by marking one of the messages representing the task and issuing a Stop Task command. The messages are then moved to a permanent storage place (a folder) defined by the user. The rule database is updated and the information concerning the task in the automation part of the system is deactivated, “unlearned” or simply removed.61

We note that the support for task management as implemented in the prototype can be employed at three different levels: adding the automatic filtering rules, thread matching and use of templates incrementally if one wishes. This is the same philosophy used in the Internet: changes are proposed in incremental steps (cf. Crocker 1993). That is, incremental changes are the best unless the new solution is easy to implement.

This ends our walk-through of the prototype. In the next chapter we evaluate the implementation of the conversation support module in the conceptual model. In other words, we test and evaluate the threading and the filtering capability of our system.

---

61. Perhaps it would be interesting to save the filtering rules together with the finished task, for later reuse. See also Future work in Chapter 12 regarding a special Task Database.
Chapter 10
Evaluation

The evaluation of our implementation of the conceptual model described in Chapter 4 is here divided into two main parts. In the first part we evaluate the thread-matching algorithm, and in the second part we evaluate the filtering rules created by RIPPER in the context of our system. The implementation was described in Chapters 8–10. We start by describing the set-up, i.e., the test subjects (persons) and the message collections used in the evaluation. Thereafter, the method used in each of the main sections mentioned above is described in detail. Then we go on to present and discuss the results from the evaluation. We sum the chapter up with conclusions.

10.1 Introduction
We evaluated the prototype from a user’s point of view. We elected to concentrate on the conversation support module in the architecture (see Figure 4.3 on p. 59), i.e., the support for threading messages, handling threads, and generating filtering rules. From a user’s viewpoint, two main questions are:

• “How well does the thread-matching algorithm perform when classifying messages into threads in our system—and compared to a manual classification?”
• “How well do the filtering rules created by RIPPER perform when classifying messages into folders in our system? A follow-up question
Chapter 10

to this was: “What type of rules are created and what can the rules tell us about the message collections?”

We were interested in both objective and subjective measures, such as counting the number of messages and threads (objective) and comparing manual classification with automatic classification of messages into threads (subjective).

10.2 The set-up

The evaluation was separated into two parts as follows:

1. In Part 1 we created message threads from a collection of messages, in order to measure the classification accuracy of the thread-matching algorithm.
2. In Part 2 we conducted a “leave-one-out” experiment involving the filtering rules to measure the classification accuracy of RIPPER in the context of our system. We also examined what type of rules RIPPER discovered for the threads in Part 1, so as to learn something more about the message collections and how RIPPER sees them.

We now describe the test subjects used, followed by the message collections and the detailed description of the two parts listed above.

10.2.1 The test subjects

As test subjects we employed persons from the department who were willing to let us use their stored e-mail communication. We presupposed that each test subject, apart from having a couple of hours to spare, also had at least a couple of different collections of not too personal e-mail, which we were allowed to examine and use in the evaluation.

In a pilot study we used the thesis author as the test subject. The test subject in the full-blown evaluation was an assistant professor, who has different functions at the department.

10.2.2 The message collections

We used the following three message collections in the evaluation of our system:

1. The first collection, which was used for the pilot study of the thread-matching algorithm, consisted of messages exchanged mainly between the thesis author and an undergraduate student doing her final thesis.
2. The second collection was borrowed from the test subject, an assistant


**Evaluation**

professor at the department. This collection consisted of messages exchanged during the arrangements of a workshop for which the test subject was the chair. It was used for evaluating the accuracy of the thread-matching algorithm.

3. The third collection was taken from the thesis author, consisting of messages on a wide range of topics, personal as well as professional communication. This collection was used to evaluate the accuracy of RIPPER’s filtering rules.

The first collection contained 66 messages in Swedish, and covered a period of 8 months. There were 6 participants in the communication represented in the first collection. A summary of the characteristics of this message collection is shown in Table 10.1.

**Table 10.1: Characteristics of the first collection of messages (the pilot study)**

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of communication (days)</td>
<td>236</td>
</tr>
<tr>
<td>Total no. of messages</td>
<td>66</td>
</tr>
<tr>
<td>No. of persons involved(^a)</td>
<td>6</td>
</tr>
<tr>
<td>No. of messages with <strong>Message-id</strong> headers(^b)</td>
<td>45</td>
</tr>
<tr>
<td>No. of messages with <strong>In-reply-to</strong> or <strong>References</strong>: headers</td>
<td>21</td>
</tr>
<tr>
<td>No. of messages with unique <strong>Subject</strong>: headers</td>
<td>34</td>
</tr>
</tbody>
</table>

\(^a\) This was determined by listing the unique e-mail addresses (receivers) found in the **From:**, **To:** and **Cc:** headers.

\(^b\) The messages that lacked a **Message-id**: header were copies of outgoing messages.

The messages without a **Message-id**: header in this collection consisted of copies of outgoing messages, originally sent by the test subject (the owner of the collection) but manually refilled from the outbox in the folder where the other messages were stored. Normally, the **Message-id**: header is added to a message by the Message Transfer Agent (MTA) (see Chapter 3).

The characteristics of the second collection are summarised in Table 10.2 on p. 198. The second collection contained 115 messages, from a period of 22 days, and involved 85 persons all in all. The messages in this collection were mainly in English, with 7 of the 115 messages in Swedish.

The second collection was actually a subset of a larger collection of messages which we got from the test subject. We selected a subset in order to limit the amount of messages and thereby the time for the evaluation.
This subset consisted of the messages from and including the announce-
ment of the (first version of the) program for the workshop up to and
including the last message just before the workshop took place.

Table 10.2: Characteristics of the second collection of messages (the
workshop arrangement)

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of communication (days)</td>
<td>22</td>
</tr>
<tr>
<td>Total no. of messages</td>
<td>115</td>
</tr>
<tr>
<td>No. of persons involved</td>
<td>85a</td>
</tr>
<tr>
<td>No. of messages with Message-id: headers</td>
<td>115</td>
</tr>
<tr>
<td>No. of messages with In-reply-to: or References: headers</td>
<td>33b</td>
</tr>
<tr>
<td>No. of messages with unique Subject: headers</td>
<td>49</td>
</tr>
</tbody>
</table>

a. One of the addresses used was a mailing list with an unknown number of
receivers, so the number of persons involved is at least this large in the col-
lection. Also, we counted the number of unique e-mail addresses, which
means that some persons may be using different e-mail addresses which are
not accounted for here.

b. There were none with both types of headers.

A closer look at the two collections described above revealed the following
characteristics. With regard to message references, in the first collection the
number of messages with In-reply-to: and/or References: headers
was 21 out of 66. However, we counted to a total of 32 messages that were
tagged with the Re: or the SV: token (the Swedish equivalence to Re:).
20 of these messages were from the test subject. The reason for the discrep-
ancy between the number of messages with In-reply-to: or Refer-
ences: headers (21) and the total number of messages with a reply token in
the Subject: header (32) is explained by a lack of functionality in the e-
mail client used by the communicating parties. The e-mail client simply did
not add In-reply-to: or References: headers to replies.

Similarly, the number of messages in the second collection with an In-
reply-to: or References: header was 33, but we counted to a total of
52 messages that were tagged with the Re: token (17 of which were from
the test subject). We also noted that an additional 6 messages were tagged
with a re: token in lower case—these had actually been created by the
test subject manually. We concluded for the second collection of messages
that the test subject’s e-mail client does not add any In-reply-to: or
References: headers to replies. A total of 52 messages, including
replies, were sent by the test subject. On the other hand, the other reply
messages in the collection all contained either an **In-reply-to:** or **References:** header, or both.

When we examined the **Subject:** header contents, we noted that a large number of them in the first collection were of a general nature and similar in content. They were referencing the title of the final thesis work, which was being performed during the communication.

In the second collection a large number of messages contained irrelevant **Subject:** headers as compared to the contents of the messages. This was due to the common use of the reply function on old messages to automatically fill in the receiver’s address in the **To:** header, a behaviour which we discussed in Chapter 3.

To summarise, the second collection’s original threads were broken for two reasons:

1. The communicating parties’ habit of using the reply function to construct a new message, in order to get the address field (the **To:** header) automatically filled in, although the new message was on a completely new topic.

2. The non-existing **In-reply-to:** and **References:** headers in the messages sent by the test subject, missing because of the e-mail client used.

The relatively large number of messages with **Message-id:** headers in the second collection, as compared to the first collection, is a side-effect of another habit of the test subject in the second collection. The test subject regularly puts his own address in the **Cc:** header on outgoing messages in order to simply get a copy of all the outgoing messages into the inbox. He does this even though the e-mail client (the User Agent; UA) already has a separate folder where all sent mail is automatically stored. In the test subject’s case, the inbox is simply used as a kind of routing central from which incoming messages are manually filed to different folders. This is a typical example of an ad hoc management of e-mail developed by a user, and a result from the limited or (to the common user) unknown functions of an e-mail client. We finally note that a good side-effect of sending a copy to oneself by using the **Cc:** header is that each message will contain a **Message-id:** header, added by the MTA—as opposed to messages stored directly in the UA’s outbox, which was the case in the first collection.

We also used a third collection of messages in the evaluation. The characteristics of this collection are summarised in Table 10.3 below. The main
CHAPTER 10

Table 10.3: Characteristics of the third collection of messages (the author’s personal collection)

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total no. of messages in collection</td>
<td>1742</td>
</tr>
<tr>
<td>Total no. of folders</td>
<td>67 (59)a</td>
</tr>
<tr>
<td>Total no. of main folders as targets</td>
<td>22b</td>
</tr>
<tr>
<td>Folders with mailing list and (or) prefixed messagesc</td>
<td>31 (30)d</td>
</tr>
<tr>
<td>Depth of folder hierarchy</td>
<td>0–2e</td>
</tr>
<tr>
<td>Mean no. of target subfolders per main folder</td>
<td>2.0</td>
</tr>
<tr>
<td>Range of no. of target subfolders per main folder</td>
<td>0–9</td>
</tr>
<tr>
<td>Mean no. of messages per folder</td>
<td>26.0 (29.5)f</td>
</tr>
<tr>
<td>Std deviation of no. of messages per folder</td>
<td>43.4 (45.1)g</td>
</tr>
<tr>
<td>Range of messages per folder</td>
<td>0–245</td>
</tr>
</tbody>
</table>

a. The number in parentheses is the number of folders that were not empty.
b. Of these, 11 contained subfolders.
c. These folders are characterised (50% or more) by group messages, such as mailing list messages (address of mailing list in To: or Cc: header), messages with a common prefix in the Subject: header, and (or) subscribed newsletters or similar, which have a common From: header. These messages represent rules that can be common to formulate as a user.
d. There was one empty folder which we knew would ordinary contain mailing list messages. The number in parentheses is excluding empty folders.
e. A value of zero means that there are no subfolders, i.e., they are the main folders (22). There was one (1) folder with a depth of 2; the rest (44) had a depth of 1.
f. The number in parentheses is the mean excluding empty folders.
g. The number in parentheses is the standard deviation excluding empty folders.

folders and relations to subfolders in the collection are presented graphically in Figure 10.1 on p. 201. This collection was used for evaluating the filtering rules produced by RIPPER and subsequently used in the context of our system. The collection was lifted from the thesis author’s messages.names file, which is the list of often used folders in our system. The file is part of the input to RIPPER (see p. 170 and p. 173).

The folders marked “X” were not part of the messages.names file and consequently not used in the evaluation. The empty folders represent folders that contained messages that were older than two months (see Section 8.3.3 on p. 170).
Figure 10.1: The folders of the third collection.

Legend:
- M = Main folder
- SF = Subfolder (followed by ID number)
- X = Placeholder (not used)
- L = Folder consisting mainly of mailing list messages
- The number within parentheses is the number of messages in the folder.
10.3 The method

Here, we describe the evaluation method used for each of the two main parts mentioned above. An overview is shown in Figure 10.2 below.

First we describe the method for evaluating the thread-matching algorithm (see Section 10.3.1 below)—this is called Part 1. Then we present the method for evaluating the filtering rules (see Section 10.3.2 on p. 208)—this we call Part 2 of the evaluation.

<table>
<thead>
<tr>
<th>Part 1: Thread-matching</th>
<th>Part 2: Filtering rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 1A: Drag-and-drop</td>
<td>Part 2A: Leave-one-out experiment</td>
</tr>
<tr>
<td>Part 1B: Re-enactment</td>
<td>Part 2B: Rules analysis</td>
</tr>
</tbody>
</table>

Figure 10.2: The two main parts of the evaluation and their contents.

10.3.1 Method for evaluating thread-matching accuracy in Part 1

This part of the evaluation of our system was considered with the creation of message threads using the first two message collections described above. The resulting threads were evaluated using two approaches, so consequently the evaluation consisted of two parts:

- **Part 1A:** A *drag-and-drop* thread-matching of the messages, where the messages were moved one-by-one into the Tasks folder
- **Part 1B:** A *re-enactment* of the exchange of messages, based on known replies (see Section 10.2.2 on p. 196), in which FLC information was included when moving messages one-by-one into the Tasks folder.

We performed a pilot study to test the evaluation method on the thread-matching algorithm before we did the full-blown evaluation. At the same time, we also checked that the messages were in chronological order. In the second collection, we actually found two pairs of messages that were in the wrong order because of different time zones in the **Date:** headers, but which—this was quite obvious in the two cases—should be put in a different order.

Since we needed target threads with which to compare the results from the thread-matching algorithm, the evaluation in Part 1 was initialised in the following way. We first asked the test subjects to *manually classify* the messages in their respective collections. To enable this, we printed each
message on a separate paper using the MP program (a PostScript pretty printer). We used the following combination of Unix commands to build the print alias:

```
alias print 'mp -m -F -s "\!*" <\!* | lp &'
```

The option `-m` in the `mp` command formats the output as an e-mail message, `-F` gives as the header who the e-mail message is from. The option `-s`, finally, gives as the footer the name of the file, which is the number of the message. The purpose of using the MP program was to give us a layout on paper very similar to what is shown on the screen of a typical e-mail client (see Figure 10.3 on p. 204).

The messages were presented in time order, one-by-one, to the test subject, who was asked to sort the messages into piles (categories or tasks), which were labelled using post-it notes. Every message had to be classified, even if they were not part of a task as interpreted by the test subject. It was possible to create links, i.e., a message was allowed to belong to several categories. However, the original copy of the message had to be stored in the category where it fitted the best. No subcategories were allowed, although these could be hinted at by using general and more specific category labels. For example, the category name “krdb” is more general than “krdb program” (two actual categories found in the second collection of messages). As mentioned above, the results of this manual classification—the threads—were used as targets for the thread-matching algorithm in both Part 1A and Part 1B.

In Part 1A we simply moved (dragged-and-dropped) each message one-by-one into the Tasks folder. Simultaneously, we logged how the messages were classified and how the threads were built by our thread-matching algorithm. We logged the ranking score and whether the suggested thread was the correct one (the target) or not. If the thread suggested by the algorithm was not correct, we also noted the ranking score and place of the closest correct message and the thread it belonged to. No FLC was employed, i.e., no message templates were present in Part 1A. The messages were however automatically tagged by the system as FLC messages with the attitude unknown. This is the default if no X-Message-Type: header is present, i.e., no message template is used.

62. Only after the evaluation did we realise that probably the Subject: header contents would have been more appropriate to put in the footer of the paper copies, instead of the message number. In this case a much shorter command is available: `mailp -F` (the MAILP program is a frontend to the MP program).
In Part 1B we re-enacted the communication that had taken place between the test subject and the other persons involved in the two collections. We assumed that every message belonged to a task. That is, whenever the test subject (the collection owner) was noted as the sender of a message, the message type and other FLC information were created and attached to the message. Also, any replies made by the test subject were handled as described on p. 185 ("person 2"). In Part 1B we also simulated that two other persons in the communication, coded as person A and person B, were using...
our system. We selected the two most frequently communicating parties in each of the two collections, in addition to the test subject. This was done in order to evaluate how FLC information in incoming messages is handled. The messages from all of these three persons in the communication were manually prepared with the corresponding FLC structure. When the messages were moved to the Tasks folder, information about the thread-matching was logged as described for Part 1A (see above).

For the purpose of Part 1B, the initialisation procedure with the manual classification described above was extended with a manual preparation procedure. The whole procedure was as follows (see also Figure 10.4):

1. Let the test subject manually classify the messages in the collection into tasks (threads) as described above.
2. Determine the message type for messages sent by the test subject and the other two persons A and B assumed to be using our system (see above).
3. Determine the task name for each message. For messages sent by the test subject, use the category from the manual classification in point 1 above. For persons A and B, use the Subject: header, except for replies made to messages sent by the test subject, and for messages where the Subject: header is obviously not relevant for the content. For the replies, keep the task name suggested by the test subject in the FLC structure.

![Figure 10.4: The procedure for Part 1B of the evaluation.](image-url)
4. For replies, note the message ID being referenced and include it in the FLC structure.
5. Fill in the rest of the FLC structure and add it as an XML attachment to the message.
6. Store the prepared messages in a separate, temporary folder along with the other messages from the other persons involved in the communication.
7. Take each message from the temporary folder and move them one-by-one to the Tasks folder. Log what happens. In this way, the communication is re-enacted, i.e., the FLC structures of the other two persons involved in the communication are also parsed by the test subject’s prototype.

The replies (cf. Section 10.2.2 on p. 196) made by the test subject were also acted out, i.e., we did actual replies when the test subject had made replies in the communication, although we did not of course send the messages. In these cases the replied-to message was inserted into a thread according to the thread-matching algorithm described in Section 8.3.1 on p. 165, and the reply message created by the test subject’s prototype was automatically stored together with the replied-to message, as described on p. 185.

As noted above, the task names for the messages from persons A and B were taken from the Subject: header, but in some obvious cases we changed the task name to something more relevant than what was available in the Subject: header. These cases included situations where the test subject had used the reply function on an old message to conveniently fill in the To: header (cf. Chapter 3). For example, in one case in the second collection the Subject: was “E-mail addresses,” but the content was really about checking that there were at least one author per paper registered for the workshop. In this case the message ID in the In-reply-to: header also gave evidence that an old message had been used for the reply.

By re-enacting the communication as described above, the messages from the test subject were updated with the appropriate FLC structure “on-the-fly” by our system—the message type, message ID, etc. We also got a mixture of FLC-structured messages and ordinary messages that the system tried to thread and create an interconnected FLC structure out of. In the first collection a total of 65 of the 66 messages were structured (prepared) using FLC. Of these messages, 20 were from the test subject and the other messages from the other two frequently communicating parties in the collection:
Evaluation

person A was responsible for 42 of these messages and person B for 3 messages. In the second collection 71 of the 115 messages were structured using FLC. Of these, 52 messages were from the test subject and the other ones from the two other selected communicating parties: 12 from person A, and 7 messages from person B.

The roles of the three persons in the communication in the first collection were as follows. The test subject was the supervisor of the final thesis worker, person A was the final thesis worker, and person B was the administrator taking care of thesis registrations and the like. In the second collection the roles of the three most frequently communicating persons were as follows. The test subject was the chair of one workshop and the local chair of a second workshop, person A was the chair of the second workshop that was co-located with the first workshop, and finally person B was the session chair of a joint session between the two workshops.

For Part 1B we also noted the number of messages from each person (person A and person B) that were replies to messages sent by the test subject. In the first collection, out of all of the replies, a total of 13 messages were replies from person A or person B to the test subject: 10 of these messages were replies from person A, and 3 messages were replies from person B—the rest (20 messages) were replies made by the test subject to person A. We noticed that out of these 13 replies only one contained a reference by message ID—the other replies only contained the reply token in the Subject: header. The message ID references was found in a References: header in the first message sent by person A. This first message reply from person A was sent using a different e-mail client, which was not subsequently used in the communication because person A had moved to another location.

With regard to person A and person B and the messages in the second collection we noted that out of all the replies in the collection, 7 messages were replies from person A to the test subject, and 3 messages from person B to the test subject. All of these replies contained either an In-reply-to: or a References: header, referencing the replied-to message.

As in Part 1A (see above), for every message moved into the Tasks folder we either accepted the suggested thread for the message in question, or selected another thread from the list that was presented—or marked the start of a new task (cf. the Threads Browser in Figure 9.7 on p. 185). We noted the method (task name, message ID, Subject: header, or body text) used by our system for finding a match, and counted the number of times that the suggested task was the correct one.
In addition to logging the results of the thread-matching algorithm, we also saved the resulting collections of messages and threads. The results were analysed using UNIX commands, AWK and shell scripts, in addition to EXCEL and STAROFFICE CALC spreadsheets. The latter were also used for displaying the results graphically.

10.3.2 Method for evaluating filtering rules accuracy in Part 2

This evaluation part was concerned with the creation of RIPPER filtering rules partly from the third collection of messages described above, and partly from the resulting collections in Part 1 above. Hence, the filtering rules were evaluated in two different ways:

• Part 2A: A leave-one-out experiment with messages from the third collection, where rules were generated and converted to the PROCMAIL format, and subsequently evaluated.

• Part 2B: An analysis of the rules created from the threads in the resulting collections in Part 1 (see Section 10.3.1 on p. 202), and comparison with the messages, threads and respective collections.

In Part 2A, two supersets were performed. In the first superset 10 random folders out of the 67 folders in the third collection (see Table 10.3 on p. 200) were used to generate the filtering rules. In the second superset 20 random folders were selected out of the 67 folders. We repeated the selection of these random folders 3 times, so that we had 3 sets of messages and folders to experiment with in each superset. In each set we experimented with two different parameter settings: including all headers from the messages in the classification process, and including only the To:, From:, and Subject: headers. The purpose of this was to see if it made any difference in the accuracy of the rules in parsing more (less) headers.

The accuracy of the filtering rules was evaluated in the context of our system. This means that the rules created for each set of messages were generated following the algorithm described in Figure 8.4 on p. 168. That is, the rules from RIPPER were converted into rules described using PROCMAIL’s syntax. We employed the leave-one-out experiment to evaluate the rules. The algorithm for this experiment is described in Figure 10.5 below. In other words, each document (message) in the corpus (message collection) was labelled according to a model built on the rest of

63. With “all headers” we mean To:, From:, Cc:, Subject:, Sender:, Date:, and including headers matching the pattern X-*: (see Section 8.3.3 on p. 170).
the e-mail messages. Each run of the experiment was logged. After each finished set we saved messages.names (the list of 10 folder names, formatted for RIPPER), the file .index_messages (a list of the messages, grouped by folders), and the folders themselves, containing the messages. The saved files were used in the analysis afterwards.

In Part 2B of evaluating the filtering rules, we returned to the two collections used in the evaluation of the thread-matching mechanism in Part 1 (see Section 10.2.2 on p. 196). We ran RIPPER on each of the two collections in the thread-matching experiment and examined the rules. That is, the targets for the classification process were the threads. By analysing the rules, we expected to learn something more about the way RIPPER sees the message collections. Furthermore, we hoped to learn something more about the message collections themselves, in addition to the characteristics of them presented earlier.

10.4 Results and discussion of the evaluation

10.4.1 Part 1A: Drag-and-drop thread-matching results

The results of the drag-and-drop thread-matching performed on the first collection of messages (the pilot study) are shown in Table 10.4 on p. 210. We note that there is a recurring task in the pilot study collection, namely the rapp-komm-x task. This task corresponds to the reviewing of different versions of the report written by the final thesis worker participating in the communication.

The results of the workshop arrangement communication in the second collection of messages are shown in Table 10.5 on p. 211. Here, the

---

**Figure 10.5:** The leave-one-out experiment.
number of messages per thread ranged from 1 to 33, with an average of 9.6 messages per thread. The length in days of a thread ranged from 1 to 22, with an average of 11.5 days per thread. As was the case in the first collection, the \text{Subject}: header was most useful for classifying messages cor-

\begin{table}[h!]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
Thread no. & Thread name & No. of msgs & Length (days) & No. of persons & No. of \text{Subject}: headers & No. of msgs correctly (incorrectly) suggested, based on... &  \\
\hline
1 & uppstart & 7 & 8 & 3 & 3 & 3 & 1 & 2 (0) & 2 (0) \\
2 & intromote & 6 & 35 & 2 & 2 & 0 & 3 & 1 (1) & 1 (1) \\
3 & planering & 3 & 10 & 3 & 2 & 0 & 0 & 0 (2) & 0 (2) \\
4 & julimote & 4 & 3 & 2 & 2 & 0 & 0 & 0 (3) & 2 (1) \\
5 & rappkomm-1 & 3 & 4 & 2 & 2 & 0 & 0 & 0 (2) & 1 (1) \\
6 & tips & 1 & 1 & 2 & 1 & - & - & - & - \\
7 & rappkomm-2 & 4 & 28 & 2 & 2 & 0 & 1 & 0 (2) & 1 (1) \\
8 & oktobermote & 9 & 10 & 2 & 3 & 0 & 0 & 0 (2) & 1 (1) \\
9 & rappkomm-3 & 3 & 11 & 2 & 2 & 0 & 0 & 0 (1) & 0 (1) \\
10 & framlagging & 5 & 1 & 3 & 4 & 0 & 0 & 1 (3) & 2 (2) \\
11 & rappkomm-4 & 7 & 37 & 2 & 6 & 0 & 1 & 0 (5) & 1 (4) \\
12 & opponent & 1 & 1 & 2 & 1 & - & - & - & - \\
13 & tryckning & 13 & 26 & 3 & 7 & 0 & 6 & 1 (5) & 1 (5) \\
\hline
\text{Mean and range per thread} & & 7.0 & 19.0 & 2.3 & 2.8 & 0.2 & 1.5 & 0.4 (2.0) & 0.9 (1.5) \\
\hline
\text{Total:} & 3 & 19 & 5 (26) & 12 (19) &
\end{tabular}
\caption{Part 1A: Drag-and-drop results of the first message collection (the pilot study)}
\end{table}

a. The number of unique, normalised \text{Subject}: headers in the thread.
b. The first message in each thread is not counted since it is by definition always correctly classified, put there by the test subject. The number in parentheses denotes the number of incorrectly classified messages. The messages classified based on matching the \text{Subject}: header and message ID, respectively, were correctly suggested all the time.


**EVALUATION**

Table 10.5: Part IA: Drag-and-drop results of the second message collection (the workshop arrangement communication)

| Thread no. | Thread name                  | No. of msgs | Length (days) | No. of persons | No. of Subject : headers | No. of msgs correctly (incorrectly) suggested, based on… |
|------------|------------------------------|-------------|---------------|----------------------|----------------------------------|
|            |                              |             |               |                      | Message ID                       | Subject | 1st attempt: Body (TF-IDF) | 2nd attempt: Body (TF-IDF) |
| 1          | kdb99 program                | 2           | 7             | 43                   | 1                                | 0 (0)   | 1 (0) | 0 (0) | 0 (0) |
| 2          | local info                   | 7           | 21            | 38                   | 6                                | 0 (0)   | 1 (0) | 2 (3) | 1 (4) |
| 3          | technical support            | 10          | 21            | 10                   | 6                                | 0 (0)   | 4 (0) | 0 (4) | 0 (4) |
| 4          | registration                 | 33          | 22            | 22                   | 14                               | 3 (2)   | 8 (2) | 3 (8) | 5 (6) |
| 5          | hotel debbie stockholm       | 1           | 1             | 2                    | 1                                | -       | -     | -     | -     |
| 6          | invited talk                 | 16          | 22            | 9                    | 8                                | 1 (0)   | 5 (3) | 1 (4) | 1 (4) |
| 7          | dl program                   | 26          | 12            | 53                   | 11                               | 7 (0)   | 3 (0) | 8 (2) | 7 (3) |
| 8          | local organisation           | 5           | 19            | 9                    | 4                                | 1 (0)   | 0 (0) | 2 (1) | 2 (1) |
| 9          | sponsor                      | 9           | 9             | 3                    | 2                                | 4 (0)   | 0 (0) | 0 (1) | 0 (1) |
| 10         | publicity                     | 2           | 2             | 2                    | 1                                | 0 (0)   | 1 (0) | 0 (0) | 0 (0) |
| 11         | other                        | 3           | 1             | 3                    | 2                                | 0 (1)   | 1 (0) | 0 (0) | 0 (0) |
| 12         | coren                        | 1           | 1             | 2                    | 1                                | -       | -     | -     | -     |

Mean and range per thread:

<table>
<thead>
<tr>
<th>Thread name</th>
<th>No. of msgs</th>
<th>Length (days)</th>
<th>No. of persons</th>
<th>No. of Subject : headers</th>
<th>No. of msgs correctly (incorrectly) suggested, based on…</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9.6 (1-18)</td>
<td>11.5 (1-25)</td>
<td>16.3 (2-53)</td>
<td>4.75 (0-11)</td>
<td>1.33 (0.0-2.60)</td>
</tr>
</tbody>
</table>

Total:

<table>
<thead>
<tr>
<th>No. of msgs correctly (incorrectly) suggested, based on…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message ID: 16 (3)</td>
</tr>
</tbody>
</table>

Second best was text matching and matching on message IDs, both 16 cases. There were errors in each of the matching algorithms in the second collection: 3 messages had to be reclassified from a previous match based on message IDs, 5 messages based on the Subject: header, and 23 on text matching (cf. Table 10.5).

Note that the sum of the number of messages (53) in the three last columns of Table 10.4 (message ID, Subject and TF-IDF) is less than the total number of messages in the collection (66). This is because firstly, at start when the Tasks folder is empty, the first message of the first task is always correctly classified by the system since there are no other messages to
match it against in the Tasks folder. Secondly, the first message of each subsequent new task, i.e., thread, is always incorrectly classified. The reason for this is that in the set-up for Part 1A of the thread-matching evaluation, our system has no way of knowing that a new task is to be started when a message is moved to the Tasks folder.64

The two Body columns in Tables 10.4 and 10.5 reflect the enhancements that we made on the text matching algorithm after a first evaluation in the pilot study. In a first attempt we actually used a different weighting of terms and formula for the ranking score than the one described in Section 8.3.1 on p. 165. We started with a straightforward TF multiplied with an IDF calculated as $1/n_k$, where $n_k$ is the number of documents containing query term $k$. This algorithm (here called TF-IDF1) turned out to be too crude for our purposes—the number of misclassified messages was too high, as compared to a random algorithm (sic)—so we repeated the drag-and-drop thread-matching evaluation with the changed algorithm on the pilot study. This showed a clear improvement.

The updated TF-IDF algorithm, based on the equations presented in Section 8.3.1 on p. 165 and here called TF-IDF2, managed to classify 7 more messages correctly (12 instead of only 5; see Table 10.4 on p. 210). We repeated the drag-and-drop thread-matching using the two variants of the TF-IDF algorithm on the second collection. The results of these two different algorithms are summarised graphically in Figure 10.5 on p. 213 for the first collection and in Figure 10.6 on p. 214 for the second collection of messages. For each case when the text matching had to be employed two bars are shown: the left one for the first, crude algorithm and the right one for the improved algorithm. The lower the rank is, the better the result is. A rank of 1 represents a correctly suggested thread for a message. The rank tends to get higher the more messages there are to parse because there are more hits to consider—this was an expected result. The horizontal axis can also be seen as a time axis.

Even though the total number of correctly and incorrectly classified messages with regard to text matching on the body stayed the same when evaluating using the second collection (see Table 10.5 on p. 211), the average rank got better. In the first attempt the average rank for a misclassified

64. If FLC had been used, as is the case in Part 1B and the evaluation of the thread-matching, the FLC structure would have contained a new conversation marker and the system would have immediately discovered that a new task should be started.
message in the second collection was 8.36, which improved to 6.45 in the second attempt.

The reason for the consistently high rank for message no. 95 in the second collection (see Figure 10.6 on p. 214) is that it is a message in English being matched against a message in Swedish. This is a well-known problem with the TF-IDF algorithm—it is language-dependent. Here a special language-independent method could be used: Latent Semantic Indexing (LSI), an extension to the standard vector retrieval method (see Berry, Dumais, & Letsche 1995). LSI takes into account the associations among terms (keywords) and documents and uses them in retrieval. LSI assumes there is a “latent” structure in the pattern of word usage across documents. By using statistical algorithms such as Singular-Value Decomposition (SVD) (Berry, Dumais, & Letsche 1995) LSI can retrieve relevant documents that do not share any words with the query.

In the first collection, the average rank of a message misclassified by the text matching algorithm improved from 9.7 using the first version (TF-IDF1) to 8.6 with the introduction of the better text matching algorithm (TF-IDF2) in the second attempt. The total percentage of correctly classified messages increased from 50.9% including the TF-IDF1 to 64.1% including the TF-IDF2 algorithm. The distribution of the three different

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure10.5.png}
\caption{The two different text matching algorithms and the first collection.}
\end{figure}
Figure 10.6: The two different text matching algorithms and the second collection.
Figure 10.7: The first collection and the distribution of the three matching algorithms over the correctly classified messages.

Figure 10.8: The second collection and the distribution of the three matching algorithms over the correctly classified messages.

matching algorithms used on the first collection is shown in Figure 10.7. The corresponding distribution for the second collection containing the workshop arrangement communication is shown in Figure 10.8 below.

Looking at the results for the first collection we note that the number of messages in a thread ranged from 1 to 13 (7 on average). A thread ranged from 1 to 37 days in length (19 days in average). Analysing the contents of the Subject: header was most useful for classifying messages correctly, followed by matching text in the message body, and matching message IDs (traditional thread-matching). One thread (no. 10) consisted of
5 messages exchanged within only 1 day. Another thread (no. 7) lasted for 28 days, with an exchange of only 4 messages. Two threads (no. 6 and no. 12) consisted of only one message each. The Subject: header was responsible for classifying 19 of the messages, while matching on the ID of messages was used for 3 messages. Successful text matching was used for 5 messages. In total 27 messages were correctly classified, while the suggested thread for 26 messages had to be changed by the test subject to a correct one. That is, for 26 messages of the 66 messages in the collection (39%) a correct matching thread was not found automatically.

The threads associated with the messages that were misclassified in the first collection based on the text in the body (see the last two columns in Table 10.4 on p. 210) are shown in Figure 10.9 on p. 217. In the figure we display the results for each variant of the text matching algorithm: TF-IDF1 and TF-IDF2, respectively. The corresponding results for the second collection are shown in Figure 10.10 on p. 218. The figure summarises the 20 references to other threads in the workshop arrangement communication made by the different versions of the TF-IDF algorithm when it misclassified messages.

In the first attempt in the pilot study collection (TF-IDF1 in Figure 10.9 on p. 217), we note that threads no. 1 (“uppstart”) and no. 3 (“planering”) seemed to be characteristic of the communication, since they were most often suggested as suitable threads: thread no. 1 was suggested 13 times, and thread no. 3 was suggested 11 times. The reason for the popularity of thread no. 1, which contained 7 messages, and thread no. 3, which contained 3 messages, could be that they both contain the most central keywords in the whole communication. This is a reasonable assumption since the threads come early in time in the collection, and especially in thread no. 1, the subject of the final thesis involved in the communication is described in detail. Thread no. 3 is dominated by a weekly report containing central keywords for the communication.

Generally, the number of misclassifications made by the TF-IDF algorithm that we used can be lowered by giving messages that are more recent in time higher weight. Obviously, the way the algorithm—especially TF-IDF1—functioned in the first collection, suggesting thread no. 1, which contained messages from May, is not a good choice when trying to classify a message from December (thread no. 13). The question is how large a “time window” should the algorithm take into account when doing the extra weighting?
Figure 10.9: The first collection and the references made by the TF-IDF1 and the TF-IDF2 algorithms, respectively, to other threads when misclassifying messages.
The thickness of the line denotes the no. of messages suggested in each thread:

- = 1 message
- = 3 messages
- = 4 messages
= 5 messages

**Figure 10.10:** The second collection and the references made by the TF-IDF1 and the TF-IDF2 algorithms, respectively, to other threads when misclassifying messages.
The range of threads was larger in the second attempt (TF-IDF2 in Figure 10.9) for the first collection, using the improved text matching algorithm (TF-IDF2): 8 different threads can be found among the suggested ones, namely threads no. 1, 2, 3, 4, 5, 7, 8, and 11. The most common thread suggested is no. 1 (8 times). Thread no. 5 was suggested 3 times, threads no. 4 and 3 one less than that each (2 times), and the other threads 1 time each.

There were also some popular threads in the second collection. The threads referenced (suggested) by the TF-IDF1 and TF-IDF2 algorithms (when misclassifying messages) are shown in Figure 10.10 on p. 218. Thread no. 7 “dl program” was popular as a reference in both attempts: it showed up 14 times in TF-IDF1 and 7 times in TF-IDF2. The reason for this may be that the messages in thread no. 7 contain information central to the whole communication, namely information about the workshop program. Thread no. 4 “registration” was also popular in the TF-IDF2 case: it was referenced 5 times. There were many references from thread no. 4 to no. 7, and vice versa, and also between threads no. 6 (“invited talk”) and no. 7—these threads contain common information about the workshop.

10.4.2 Part 1B: Re-enactment of Communication Results

In Part 1B we re-enacted the communication as if the test subject and two other persons involved in the communication were using a copy of our system. The results are shown in Table 10.6 on p. 221 for the first collection and Table 10.7 on p. 222 for the second collection. For the sake of overview we recapitulate in the tables some of the information presented earlier.

The threaded outcome of the re-enacted communication was compared to the manual classification performed by the test subject. We noted the times when the text matching (ranking) algorithm had to be used, along with the times it succeeded as well as failed.

The distribution of the use of the different matching algorithms (reply, message ID, Subject: header, and TF-IDF) in the thread-matching algorithm is shown in Figure 10.11 for the first collection and Figure 10.12 on p. 220 for the second collection. The matching based on reply is simply the automatic association of replies made by the test subject with the corresponding replied-to message (cf. the working example on p. 185). In the first collection a large share of the messages had to be processed by the TF-IDF algorithm so as to find a matching thread.

The analysis of the first two attempts to thread the second collection by simple drag-and-drop (see Section 10.4.1 on p. 209) gave us ideas on how to improve the text matching algorithm based on TF-IDF even more.
Figure 10.11: The first collection and the distribution of the different matching algorithms in the thread-matching algorithm.

Figure 10.12: The second collection and the distribution of the different matching algorithms in the thread-matching algorithm.
Table 10.6: Part IB: Re-enactment of communication results in the first collection

<table>
<thead>
<tr>
<th>Thread no.</th>
<th>Thread name</th>
<th>No. of msgs</th>
<th>Length (days)</th>
<th>No. of pers.</th>
<th>No. of subj.-header</th>
<th>No. of msg correctly (incorrectly) suggested, based on...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reply</td>
</tr>
<tr>
<td>1</td>
<td>uppstart</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3 (0)</td>
</tr>
<tr>
<td>2</td>
<td>intromote</td>
<td>6</td>
<td>35</td>
<td>2</td>
<td>2</td>
<td>4 (0)</td>
</tr>
<tr>
<td>3</td>
<td>planering</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>1 (0)</td>
</tr>
<tr>
<td>4</td>
<td>julimote</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2 (0)</td>
</tr>
<tr>
<td>5</td>
<td>rappkomm-1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1 (0)</td>
</tr>
<tr>
<td>6</td>
<td>tips</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>rappkomm-2</td>
<td>4</td>
<td>28</td>
<td>2</td>
<td>2</td>
<td>2 (0)</td>
</tr>
<tr>
<td>8</td>
<td>oknobermote</td>
<td>9</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>4 (0)</td>
</tr>
<tr>
<td>9</td>
<td>rappkomm-3</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>2</td>
<td>0 (0)</td>
</tr>
<tr>
<td>10</td>
<td>framlagning</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>0 (0)</td>
</tr>
<tr>
<td>11</td>
<td>rappkomm-4</td>
<td>7</td>
<td>37</td>
<td>2</td>
<td>6</td>
<td>0 (0)</td>
</tr>
<tr>
<td>12</td>
<td>opponent</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>tryckning</td>
<td>13</td>
<td>26</td>
<td>3</td>
<td>7</td>
<td>1 (0)</td>
</tr>
</tbody>
</table>

Mean and range per thread

<table>
<thead>
<tr>
<th></th>
<th>7.0 (0)</th>
<th>19.0 (1)</th>
<th>23 (0)</th>
<th>28 (0)</th>
<th>1.4 (0)</th>
<th>0.23 (0)</th>
<th>0.85 (9)</th>
<th>0.46 (1.4)</th>
<th>0.54 (1.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–13</td>
<td>1–37</td>
<td>2–3</td>
<td>1–7</td>
<td>0–4 (0)</td>
<td>0–3 (0)</td>
<td>0–6 (0)</td>
<td>0–2 (0–5)</td>
<td>0–2 (0–4)</td>
<td></td>
</tr>
</tbody>
</table>

Total: 18 (0) 3 (0) 11 (0) 6 (18) 7 (17)

a. The first message in each thread is always correctly classified because it is put there by the test subject.
Table 10.7: Re-enactment of communication results in the second collection

<table>
<thead>
<tr>
<th>Thread no.</th>
<th>Thread name</th>
<th>No. of msgs</th>
<th>Length (days)</th>
<th>No. of pers.</th>
<th>No. of Subject:headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>krdb99 program</td>
<td>2</td>
<td>7</td>
<td>43</td>
<td>1 (0) 0 (0) 0 (0) 0 (0)</td>
</tr>
<tr>
<td>2</td>
<td>local info</td>
<td>7</td>
<td>21</td>
<td>38</td>
<td>6 (0) 0 (0) 1 (4) 3 (2)</td>
</tr>
<tr>
<td>3</td>
<td>technical support</td>
<td>10</td>
<td>21</td>
<td>10</td>
<td>6 (1) 0 (0) 4 (0) 0 (4) 2 (2)</td>
</tr>
<tr>
<td>4</td>
<td>registration</td>
<td>33</td>
<td>22</td>
<td>22</td>
<td>14 (1) 3 (2) 8 (2) 5 (6) 3 (8)</td>
</tr>
<tr>
<td>5</td>
<td>hotel debbie stockholm</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1 – – – – –</td>
</tr>
<tr>
<td>6</td>
<td>invited talk</td>
<td>16</td>
<td>22</td>
<td>9</td>
<td>8 (0) 1 (0) 5 (3) 1 (4) 1 (4)</td>
</tr>
<tr>
<td>7</td>
<td>dl program</td>
<td>26</td>
<td>12</td>
<td>53</td>
<td>11 (0) 7 (0) 3 (0) 7 (3) 8 (2)</td>
</tr>
<tr>
<td>8</td>
<td>local organisation</td>
<td>5</td>
<td>19</td>
<td>9</td>
<td>4 (0) 0 (0) 0 (0) 2 (1) 2 (1)</td>
</tr>
<tr>
<td>9</td>
<td>sponsor</td>
<td>9</td>
<td>9</td>
<td>3</td>
<td>2 (0) 4 (0) 0 (0) 0 (1) 0 (1)</td>
</tr>
<tr>
<td>10</td>
<td>publicity</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1 (0) 0 (0) 1 (0) 0 (0) 0 (0)</td>
</tr>
<tr>
<td>11</td>
<td>other</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0 (1) 1 (0) 0 (0) 0 (0)</td>
</tr>
</tbody>
</table>

Mean and range per thread

|                |               |               |               |               |
|----------------|---------------|---------------|---------------|               |
|                | mean          | range         | mean          | range         |
| Length (days)  | 9.6           | 1–33          | 11.5          | 1–22          |
| No. of Subject:headers | 16.3 | 2–53 | 4.75 | 1–14 |
| No. of Pers.   | 1.17 (0.17)   | 0–5 (0–1)     | 1.33 (0.25)   | 0–7 (0–2)     |
| No. of msgs    | 2.00 (0.42)   | 0–8 (0–3)     | 1.33 (1.92)   | 0–6 (0–6)     |
| Total          | 14 (2)        | 16 (3)        | 24 (5)        | 16 (23)       |

Note: The first message in each thread is always correctly classified because it is put there by the test subject.
We decided to change the TF-IDF2 variant slightly by including the keywords (tokens) from the To:, From:, and Cc: headers as well, along with the keywords from the body of the message, in the calculation of the ranking score. The TF-IDF3 in the figures and tables in this section denotes the final version with these enhancements. For the first collection the percentage of correctly classified messages was 73.6% including TF-IDF3, versus 71.7% when including TF-IDF2 instead. That is, a total of 39 messages out of the 53 messages to be classified in the first collection (66 minus the 13 messages that started each thread) were correctly classified by a combination of the four matching algorithms when TF-IDF3 was used. The distribution of the different matching algorithms over these messages is shown in Figure 10.13. For the second collection the percentage of correctly classified messages was 70.9% including TF-IDF3, versus 68.0% when including TF-IDF2 instead. In the TF-IDF3 case this means that a total of 73 messages out of the 103 messages (115 minus the 12 messages that started a new thread) were actually classified correctly. The distribution of these messages over the different matching algorithms is shown in Figure 10.14 on p. 224 for the second collection.

As stated earlier, the test subject was, in the manual classification of the collection, allowed to create links between messages, to denote messages that should belong to several threads. In the first collection no links were employed by the test subject. In the second collection the test subject had
created links from three messages. There was a dual link from one message in thread no. 4 “registration” to thread no. 1 “krdb99 program” as well as thread no. 7 “dl program.” Also, there was a link from another message in thread no. 4 to thread no. 1 “kdrb99 program.” Finally, a link had been created between a message in thread no. 6 “invited talk” and threads no. 5 “hotel debbie stockholm” and no. 4 “registration.” Our hypothesis was that the links could give us a clue to messages that were hard to classify. Our system managed to classify all but the last message containing links correctly into one of the threads involved in the links. We also examined whether the links could give us a clue to threads that were similar in nature in some sense. This would have meant that any misclassified messages in the threads involved should have been classified as belonging to one of the other linked threads. There was no evidence of this.

The misclassifications in the second collection (see Table 10.2 on p. 198) based on the Subject: header were due to some of the Subject: headers being very general and commonly used in the communication, such as “krdb” and “dl99.” Our (greedy) algorithm matched on Subject: headers before trying to use TF-IDF. The latter is the more general matching algorithm since it also examines the message body.

As noted in the description of the message collection in Section 10.2.2 on p. 196, there was a large number of messages containing irrelevant Subject: headers in the second collection of messages—irrelevant with regard to the message body contents. This was due to the common use of
EVALUATION

the reply function on previously received messages to automatically fill in the receiver’s address in the To: header when composing a new message. The reason for this behaviour is that the users typically keep a copy of a message from all people that they communicate with, even though they may know the addresses. It is then easier to use the reply function to get the address than to use the address book of the e-mail client. This user behaviour affected the results of the thread-matching in our system. A solution to this problem is to improve the address book functionality. Bälter (1998) suggests a program that automatically scans all messages and collects the addresses. The EXMH e-mail client has this functionality. Moreover, in EXMH a partial name or address can be typed in the To:, Cc:, Bcc:, or Dcc: headers and Control-Tab pressed to fill in the rest of (expand) the address automatically—if the address is ambiguous a list of alternatives is presented to the user.

In the second collection, there were two misclassifications based on reply matching, i.e., automatically moving the reply message to the same thread as the replied-to message. In these two cases the test subject had in the manual classification stage classified each of the two replies as belonging to a separate task than the replied-to messages. The FLC structure of this kind of messages will contain a reference to a message in the correct thread, as well as a reference to the original replied-to message. However, in the header of the message and available to Internet e-mail clients, only the former reference above is made visible by our system. This is in the form of a References: header with the corresponding message ID, referencing the message in the correct thread that our system has located. In the context of threads, this reference can be said to be the local referencing of messages, with the purpose of keeping a thread together. The reference to the replied-to message—and which is also saved in the FLC structure—is what “traditional” threading uses. We call this latter the global referencing of messages—it is global when compared to a thread where local referencing is employed.

We also logged the occasions when the task name specified by the test subject was employed by the thread-matching algorithm. As stated in

65. The Dcc: (Distribution Carbon Copies) header in EXMH functions as the Bcc: (Blind Carbon Copies) header, with the difference that there is no "-------Blind-Carbon-Copy" wrapper around the message.

66. In FLC, we use the tags <referencing> and <respondingTo>, respectively, for these two types of references in the XML-encoding of the FLC structure.
Section 10.3.1 on p. 202, we assumed that whenever any of the other two persons also using a copy of the prototype replied to (or in any way used) a message received from the test subject, they should keep the task name already in the FLC structure. The results are listed in Table 10.8 for the first collection and Table 10.9 on p. 227 for the second collection. In the

Table 10.8: The use of task name in the first collection

<table>
<thead>
<tr>
<th>Thread no.</th>
<th>Thread (task) name</th>
<th>Total no. of msgs per thread</th>
<th>No. of msgs matched on task name</th>
<th>Could otherwise be matched on…</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>upstart</td>
<td>7</td>
<td>1</td>
<td>subject (1)</td>
</tr>
<tr>
<td>4</td>
<td>julimote</td>
<td>4</td>
<td>1</td>
<td>TF-IDF (1)b</td>
</tr>
<tr>
<td>8</td>
<td>oktobermote</td>
<td>9</td>
<td>3</td>
<td>subject (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TF-IDF (1)</td>
</tr>
<tr>
<td>9</td>
<td>rappkomm-3</td>
<td>3</td>
<td>1</td>
<td>subject (1)</td>
</tr>
<tr>
<td>10</td>
<td>framlaggnng</td>
<td>5</td>
<td>1</td>
<td>TF-IDF (1)</td>
</tr>
<tr>
<td>11</td>
<td>rappkomm-4</td>
<td>7</td>
<td>1</td>
<td>subject (1)</td>
</tr>
<tr>
<td>13</td>
<td>tryckning</td>
<td>13</td>
<td>3</td>
<td>subject (3)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>48</strong></td>
<td><strong>11</strong></td>
<td>subject (8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TF-IDF (3)</td>
</tr>
</tbody>
</table>

a. The number in parentheses is the no. of messages matched in each algorithm.
b. This message was not correctly classified by the TF-IDF algorithm, which means that the use of task name avoided this.

second collection we see that the use of the task name to thread messages corresponds roughly to matching messages on the message IDs. Here, 11 out of the 13 messages containing a task name specified could also have been classified using the message IDs in the messages. The two cases where other methods would have been needed in the second collection were threads 1 and 4:

• The two messages in thread no. 1 were two new messages sent by the same person, naturally with no related message IDs, so they were matched based on their Subject: headers.
The original message also belonging in thread no. 4 was missing from the collection, so no previously parsed, matching message ID existed. Matching based on TF-IDF was the last resort here.

In the first collection, matching messages based on the Subject: header corresponds roughly to matching on task name in the FLC structure (8 messages out of 11). The reason for this is that the messages from the test subject are copies of outgoing messages without message ID information, but with a Re: token in the Subject: header to denote message that should belong together.

In effect the task name becomes a way of naming threads and doing this locally. Thus, threads can be handled by name. However, it is also possible to use task names globally. If a group of communicating parties agrees upon common task names, the threads will be created based on these names rather naturally. This gives the individual less freedom in the choice of task names. Also, two persons may have different views of what the tasks are. For example, in the workshop arrangement collection we assumed person A sent a message with the task name “authors registered.” The test subject does not have any task name of that type, but instead uses the more general task name “registration.” Task names can be seen as sub-categories of a folder, which gives the user another way of storing, retrieving, and viewing information in e-mail.

---

**Table 10.9: The use of task name in second collection**

<table>
<thead>
<tr>
<th>Thread no.</th>
<th>Thread (task) name</th>
<th>Total no. of msgs per thread</th>
<th>No. of msgs matched on task name</th>
<th>Could otherwise be matched on…a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>krdb99 program</td>
<td>2</td>
<td>1</td>
<td>subject (1)</td>
</tr>
<tr>
<td>4</td>
<td>registration</td>
<td>33</td>
<td>2</td>
<td>msg ID (1), TF-IDF (1)</td>
</tr>
<tr>
<td>6</td>
<td>invited talk</td>
<td>16</td>
<td>1</td>
<td>msg ID (1)</td>
</tr>
<tr>
<td>7</td>
<td>dl program</td>
<td>26</td>
<td>5</td>
<td>msg ID (5)</td>
</tr>
<tr>
<td>9</td>
<td>sponsor</td>
<td>9</td>
<td>4</td>
<td>msg ID (4)</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td></td>
<td><strong>86</strong></td>
<td><strong>13</strong></td>
<td>msg ID (10), subject (2),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TD-IDF (1)</td>
</tr>
</tbody>
</table>

a. The number in parentheses is the no. of messages matched in each algorithm.

- The original message also belonging in thread no. 4 was missing from the collection, so no previously parsed, matching message ID existed.
Some of the messages in the second collection—4 of them—were in Swedish, which gave as a result that the text matching algorithm prioritised the other Swedish messages higher than the English ones when trying to find a match to a Swedish message. We discussed this language-dependence of the TF-IDF algorithm on p. 213.

Because we had made an improvement to the text matching algorithm we also investigated the changes and improvements in thread referencing and rank of the messages across the (now) three different TF-IDF algorithms.

The associations made by TF-IDF3 between the misclassified messages and their correct threads in the first collection of messages are shown in Figure 10.15 (cf. Figure 10.9 on p. 217). The corresponding associations for the second collection are shown in Figure 10.16 on p. 229.

In the first collection the TF-IDF3 managed to classify one more message correctly (cf. Table 10.6 on p. 221). With regard to the earlier results with TF-IDF1 and TF-IDF2, the popularity of threads no. 1 (referenced 5 times) and no. 3 (referenced 4 times) persisted (see Figure 10.15), but with lower intensity. In the second collection TF-IDF3 managed to classify 3 more messages correctly compared to TF-IDF1 and TF-IDF2. The references between threads no. 4 and no. 7 became fewer. The association

![Figure 10.15: The first collection and references made by the final (TF-IDF3) algorithm to other threads when misclassifying messages.](image-url)
between threads no. 6 ("invited talk") and no. 7 ("dl program") remained strong. Overall the number of references per thread became lower: from a maximum of 5 (using TF-IDF1) and 3 (using TF-IDF2) to a maximum of 2 per thread.

The results for the rank of messages are summarised in Figure 10.17 on p. 230 for the first collection and in Figure 10.18 on p. 232 for the second collection. In the figure, a low rank means a good result—and a rank of 1 denotes a correctly suggested thread for a message that was filed into the Tasks folder. The "1st attempt" in the figure represents the use of the crude algorithm of simply multiplying TF with IDF. Both "2nd attempt" and "3rd attempt" represent the usage of the algorithm presented in Section 8.3.1 on p. 165, but with a difference in the extraction of keywords. In the second attempt we parsed for keywords in the text body only, up to but not including any attachments. In the third attempt we added the parsing of the To:, From:, and Cc: headers in the algorithm. Looking at the first collection (see Figure 10.17 on p. 230), the total number of misclassified messages using text matching did not get any lower with the TF-IDF3 (20 messages) compared to TF-IDF2 (19 messages). However, the average rank of a message got better (lower): the rank of a message classified with the three versions of the text matching algorithm improved from 9.7

![Figure 10.16: The second collection and references made by the final (TF-IDF3) algorithm to other threads when misclassifying messages.](image_url)
in the first attempt to 8.6 in the second attempt, and to 7.1 in the third attempt. For the second collection the corresponding average ranks were 8.36 in the first attempt, 6.45 in the second attempt, and 5.85 in the third attempt (see Figure 10.18 on p. 232).

The usage of TF-IDF to match a message with other relevant messages/threads is motivated by the mechanism of lexical repetition, i.e. the repeating of words in linked parts of a discourse (cf. Lewis & Knowles 1997). In the second collection we found that 48.7% (19 messages) of the 39 messages processed by the TF-IDF3 algorithm were automatically associated with the correct thread (see Figure 10.12 on p. 220). The 39 messages passed to TF-IDF3 represent 37.9% of all of the messages needed to be classified in the second collection. For the first collection the percentage was 29.2% (7 messages) of 24 messages passed to TF-IDF3, which is 45.3% of all of the messages needed to be classified.

The results are promising. With a simple text-matching algorithm it is possible to enhance the threading mechanism. Even adding only matching on Subject: headers improves the results. As a matter of fact, the latter is a good indicator of a matching thread when the message ID is not present.
Figure 10.18: The second collection and the ranks made by the three different text matching algorithms.
Adding FLC gives us a container for task information, e.g., the name of the task. The task name also becomes the name of the thread.

10.4.3 PART 2: FILTERING RULES RESULTS

The 10 random folders of Sets 1–3 and the 20 random folders of Sets A–C, all taken from the third collection, are listed in Table 10.10 below. We also generated rules for the whole third collection, which is denoted “Collection” in the table. The folder names in the table refer to Figure 10.1 on p. 201.

In Part 2A, for each set of messages we noted the total number of messages, the mean number of messages per folder, and the range of the number of messages in each folder. Furthermore, we also examined which folders contained a majority (50% or more) of mailing list messages, and therefore had, e.g., a prefix for identifying the list in the Subject: header contents. The classification target’s type of folder was also noted, i.e., if it was a main folder or a subfolder.

Table 10.10: The contents of the filtering sets

<table>
<thead>
<tr>
<th>Foldersa</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>M2, M7, M11, SF3, SF7, SF18, SF21, SF28, SF30, SF42</td>
</tr>
<tr>
<td>Set 2</td>
<td>M10, M11, M12, SF2, SF14, SF21, SF24, SF30, SF36, SF44</td>
</tr>
<tr>
<td>Set 3</td>
<td>M1, M13, SF2, SF11, SF14, SF18, SF19, SF25, SF42 SF43</td>
</tr>
<tr>
<td>Set A</td>
<td>M2, M5, M7, M12, M13, M14, M19, M20, SF11, SF19, SF20, SF24, SF26, SF27, SF29, SF30, SF31, SF37, SF41, SF44</td>
</tr>
<tr>
<td>Set B</td>
<td>M2, M5, M9, M12, M14, M15, M17, M19, SF2, SF5, SF6, SF10, SF12, SF21, SF31, SF33, SF34, SF35, SF36, SF38</td>
</tr>
<tr>
<td>Set C</td>
<td>M1, M4, M7, M9, M11, M12, M19, M20, SF1, SF3, SF4, SF12, SF17, SF19, SF25, SF28, SF29, SF33, SF44, SF45</td>
</tr>
<tr>
<td>Collection</td>
<td>All non-empty folders</td>
</tr>
</tbody>
</table>

a. The coding refers to Figure 10.1 on p. 201.

Figure 10.19 gives an overview of the accuracy results. Generally, the classification accuracy is high for Sets 1–3. An accuracy of 83.0% (the worst result in the experiment) and higher may well satisfy an average user’s need. Segal and Kephart (2000), in their SWIFTFILE e-mail system, consider an accuracy in the interval of 53–76% as unacceptable for automated classification. In light of this, the accuracy achieved with the help of RIPPER in the context of our system is likely to be acceptable. With
regard to the parameter settings, in Sets 1–3 the “All headers” approach is favoured, but in Sets A–B the “From:, To:, Subject:” approach seems to be better. The reason for this can be that the X-headers included in “All headers” introduce too much noise when the number of messages grows—the headers are not relevant.

Table 10.11 on p. 235 summarises the characteristics of the sets used in Part 2A. “Collection” in the table is the whole third collection. The characteristics for the rules generated by RIPPER for each of the sets are also shown in Table 10.11. It is surprising that the difference in rule generation time is as large as it is between Set 1 and Set 2 (1.74 s vs. 5.08 s in the “All headers” approach). The number of conditions for the rules is high for Set 2: 107.9 in average, compared to 57.4 in average for Set 1. For that matter, the other characteristics of Sets 1 and 2 are rather similar, e.g. the number of messages is not so different (356 vs. 389).

An example set of rules generated for Set 1 is shown in Figure 10.20 on p. 234. The rules for Set 2 are shown in Figure 10.21 on p. 236. In the examples, Set 1 has 53 rules and 112 conditions, while Set 2 has 25 rules and 51 conditions. In the figures, the rules have been numbered. Following the rule number, the left-hand side of the :- token contains the folder name.

**Figure 10.19:** The filtering accuracy of RIPPER in our system.
Table 10.11: The characteristics of the sets and the generated rules in the filtering experiment.

<table>
<thead>
<tr>
<th>Set</th>
<th>Total no. of msgs</th>
<th>Total no. of (non-empty) folders</th>
<th>No. of target main folders</th>
<th>No. of mailing list/prefix msg folders</th>
<th>Mean (and std dev.) no. of rules</th>
<th>Mean (and std dev.) no. of rule conditions</th>
<th>Mean (and std dev.) time for rule generation (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>356</td>
<td>10</td>
<td>3</td>
<td>4</td>
<td>35.6</td>
<td>58.9</td>
<td>1–193 27.5 (4.00) 57.4 (10.2) 1.74 (0.33) 37.9 (2.61) 77.6 (6.64) 1.68 (0.15)</td>
</tr>
<tr>
<td>Set 2</td>
<td>389</td>
<td>10</td>
<td>3</td>
<td>5</td>
<td>38.9</td>
<td>59.4</td>
<td>1–193 50.7 (2.92) 107.9 (6.25) 5.08 (0.62) 55.1 (1.41) 112.6 (3.12) 4.23 (0.38)</td>
</tr>
<tr>
<td>Set 3</td>
<td>309</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>30.9</td>
<td>29.5</td>
<td>1–96 33.2 (2.98) 69.2 (6.21) 1.71 (0.32) 48.7 (4.52) 99.1 (10.5) 2.25 (0.61)</td>
</tr>
<tr>
<td>Set A</td>
<td>330</td>
<td>20</td>
<td>8</td>
<td>7</td>
<td>16.5</td>
<td>17.1</td>
<td>1–78 67.5 (3.90) 141.9 (8.46) 3.05 (0.31) 74.5 (1.42) 152.6 (2.93) 2.07 (0.13)</td>
</tr>
<tr>
<td>Set B</td>
<td>640</td>
<td>20</td>
<td>8</td>
<td>11</td>
<td>32.0</td>
<td>55.5</td>
<td>1–245 73.5 (4.32) 157.8 (10.10) 8.64 (0.93) 78.4 (4.98) 162.9 (9.99) 5.83 (0.62)</td>
</tr>
<tr>
<td>Set C</td>
<td>933</td>
<td>20</td>
<td>8</td>
<td>10</td>
<td>46.7</td>
<td>69.7</td>
<td>1–245 107.9 (9.58) 238.9 (21.61) 30.1 (5.93) 141.0 (12.20) 298.0 (28.1) 27.2 (5.60)</td>
</tr>
<tr>
<td>Collection</td>
<td>1742</td>
<td>59</td>
<td>11</td>
<td>30</td>
<td>29.5</td>
<td>48.1</td>
<td>1–245 220.1 (12.50) 474.9 (28.27) 90.0 (12.22) 258.7 (12.40) 545.8 (27.2) 66.6 (11.0)</td>
</tr>
</tbody>
</table>

Note: The time was calculated for each round of the leave-one-out experiment.
Evaluation

The right-hand side lists the attributes and corresponding values that the rule is triggered on. The attributes and their values are separated by an underscore character ("_") and preceded by the token WORDS ~. The numbers in parentheses at the end of each line denote how many messages the rule covers for the folder in question, and how many other messages in other folders are also covered by the rule.

With regard to Set 2, there are two folders that dominate the rule set (see Figure 10.21 on p. 236), namely folders M11 and M12. Consequently, they seem difficult to characterise. M11 has previously been characterised as a mailing list folder (see Figure 10.1 on p. 201) but as much as 50% of its messages seem to lack much in common.

The sets created for the second superset (Sets A–B) were all different in many aspects. Set A was the smallest with 330 messages, Set B the second largest with 640 messages, and Set C the largest with 933 messages. The accuracy results (see Figure 10.19 above) seem congruent with the results of Rennie (2000) in the evaluation of the e-mail filtering system IFILE. IFILE is based on a Naive Bayesian machine learning algorithm. An advantage of using a decision tree algorithm such as RIPPER is that it displays the rules in clear text. In addition, RIPPER generates good rules from the seemingly small amounts of data available in our test collections.

Figure 10.20: An example rule set for Set 1.
Regarding the experiment in Part 2B, the rules discovered by RIPPER for the pilot study test collection are listed in Figure 10.22 on p. 237. For the 66 messages and 13 threads in the pilot study test collection RIPPER thus generated 18 rules, with thread no. 2 (intrormote) treated as the default folder.

For example, rule 8 in the figure tells us that RIPPER has discovered a rule for classifying four of the messages into the rappkomm-1 thread based on the contents of the Date: header, which should contain the string Jul.

\[
\text{"M10" := WORDS = subject_alumni (17/0).} \\
\text{"M10" := WORDS = \"x-groups-return-groups\" (1/0).} \\
\text{"M1/SF21" := WORDS = \"x-mailer_listmanager\" (10/0).} \\
\text{"M1/SF21" := WORDS = to_alfa (5/1).} \\
\text{"M1/SF21" := WORDS = from_anders (6/0).} \\
\text{"M1/SF44" := WORDS = from_liu republican (1/0).} \\
\text{"M1/SF44" := WORDS = to_listserv (1/0).} \\
\text{"M4/SF24" := WORDS = subject_rappport (9/0).} \\
\text{"M4/SF24" := WORDS = \"x-sender_krist\" (7/0).} \\
\text{"M4/SF24" := WORDS = from_jussi (1/0).} \\
\text{"M4/SF24" := WORDS = subject_note (4/0).} \\
\text{"M6/SF36" := WORDS = subject_pilotmgr (44/0).} \\
\text{"M6/SF36" := WORDS = \"x-authentication-warning_moshpit\" (35/0).} \\
\text{"M11" := WORDS = subject_lai (106/0).} \\
\text{"M11" := WORDS = from_svechek (21/0).} \\
\text{"M11" := \"x-image-url\" http://www.ida.liu.se/~juhta/gif/juha.gif, WORDS = to_andreas (12/0).} \\
\text{"M11" := WORDS = \"x-mailer-express\", WORDS = \"x-mailer_2919\" (16/0).} \\
\text{"M11" := WORDS = from_student (26/0).} \\
\text{"M11" := WORDS = from_fitness (8/0).} \\
\text{"M11" := from_johab, WORDS = to_student (18/0).} \\
\text{"M11" := from_johab, WORDS = to_passinfo (5/0).} \\
\text{"M11" := to_lai (24/0).} \\
\text{"M11" := subject_bottling (3/0).} \\
\text{"M11" := subject_step (9/0).} \\
\text{"M11" := from_webedebeat (2/0).} \\
\text{"M11" := to_guru (5/0).} \\
\text{"M11" := to_instructor (40/0).} \\
\text{"M11" := to_info (1/0).} \\
\text{"M11" := to_013239686 (5/0).} \\
\text{"M11" := from_granlund (2/0).} \\
\text{"M11" := subject_spinning (1/0).} \\
\text{"M11" := subject_address (2/0).} \\
\text{"M11" := to_mattias (7/0).} \\
\text{"M11" := to_013239686 (2/0).} \\
\text{"M12" := from_nahid (9/0).} \\
\text{"M12" := \"x-mailer-express\", WORDS = date_jun (22/1).} \\
\text{"M12" := from_nahid (13/0).} \\
\text{"M12" := to_johab (8/1).} \\
\text{"M12" := subject_send (5/0).} \\
\text{"M12" := from_johab, WORDS = subject_meeting (3/0).} \\
\text{"M12" := subject_fetching (5/0).} \\
\text{"M12" := to_marcello (4/0).} \\
\text{"M12" := to_iddena (7/0).} \\
\text{"M12" := to_steve (2/1).} \\
\text{"M12" := to_013239686 (1/0).} \\
\text{"M12" := from_order (1/0).} \\
\text{"M12" := to_patrik (1/0).} \\
\text{"M12" := to_jokeli (1/1).} \\
\text{"M12" := from_order (3/0).} \\
\text{"M12" := to_rosgr (2/1).} \\
\text{"M12" := to_almut (1/0).} \\
\text{"M12" := from_order (1/0).} \\
\text{"M12" := to_x01denma (7/0).} \\
\text{"M12" := to_013239686 (0/0).} \\
\text{resolution=best.} 

----------------------------------------------------------------------

Figure 10.21: An example rule set for Set 2

Regarding the experiment in Part 2B, the rules discovered by RIPPER for the pilot study test collection are listed in Figure 10.22 on p. 237. For the 66 messages and 13 threads in the pilot study test collection RIPPER thus generated 18 rules, with thread no. 2 (intrormote) treated as the default folder.

For example, rule 8 in the figure tells us that RIPPER has discovered a rule for classifying four of the messages into the rappkomm-1 thread based on the contents of the Date: header, which should contain the string jul.
and the contents of the Subject: header, which should contain the value enhanced. The (3/1) at the end of the row means that the rule covers 3 messages in the rappkomm-1 thread, but also 1 message in another thread—it is the thread named julimote which actually contains a message with jul in the Date: header and enhanced in the Subject: header.

When we look closer at the rules in Figure 10.22 we see that many of them are based on the Date: header and its values. Should we use these categories for a longer period of time, the rules would also have to change correspondingly when RIPPER is run on the new parsed data.

When a human user manually constructs filtering rules, she typically bases them on the contents of the To:, From: and Subject: headers. For example, to sort all messages sent to a mailing list addressed to induc-tive@listserv.unb.ca a suitable rule would look as follows, using RIPPER’s rule syntax:

```
inductive IF WORDS ~ to_inductive .
```

Figure 10.22: The rules discovered by RIPPER for the first collection.
CHAPTER 10

That is, if the \textit{To:} header contains the word \textit{inductive}, then the message should be classified as belonging to the folder \textit{inductive}. We used RIPPER on the same mailing list and got the following rule:

\texttt{inductive IF WORDS ~ x-received\_from}.

In other words, if the \texttt{X-received:} header contains the word \textit{from}, then the message should be sorted into the \textit{inductive} folder. This rule also works fine—the only difference is that the first one was discovered by the human user and the other one by the machine.

In Figure 10.22 we also see an example of the problem with rule order (see Chapter 6): a message containing \textit{jul} in the \texttt{Date:} header would here match both rules 7 and 8, but in the current implementation of the filtering system only rule 7 (the first in order) will trigger on the message. This may or may not be the intended result.

The rules discovered for the second collection, the workshop arrangements communication, are shown in Figure 10.23 on p. 239. Here we have 26 rules for the 115 messages in 12 threads. We note that many of the rules for the second collection use the \texttt{Subject:} header contents to classify messages (11 of 26 rules). The default rule classifies messages to the \texttt{registration} thread.

10.5 Conclusions of the evaluation

Generally, the evaluation of an e-mail tool, using collections of e-mail, is difficult to perform since collections of e-mail messages tend to be too personal to use or to publish.

The choice of subjects (test persons) freed us from additional overhead in preparing the prototype of the system. We have used test subjects from the university world. University employees often have a lot of more external communication. This introduces some interesting problems with distorted national characters because of incompatible systems—problems that the prototype has to be able to handle. There is also a greater diversity of computer types and e-mail tools used at a university, introducing variable types of attachments, and—as we have seen—different interpretations of the e-mail standards with “missing” headers, \textit{etc}. With regard to the conceptual model, the choice of test subjects is not relevant—the model is meant to handle all different kinds of users.

An interesting thing is that by making it possible to name threads—selecting New Task and inputting a task name—we can classify individual messages within a folder. This classification is regardless of the \texttt{Subject:}.
header contents and the name of the folder. In other words, the user now has a tool for classifying parts of a discussion into other categories, although the Subject: header is the same. Moreover, in the Tasks folder the threads are made visible to the user—she is presented with the names of the threads and also a suggestion of an appropriate thread for a given message.

Being able to name threads also means being able to handle threads by name. This gives us another dimension for information retrieval tasks: the user can search for threads sorted by and having a certain (type of) name.

In our evaluation we did not take into account any annotations of telephone calls and the like that a user may have wanted to store together with the messages in a communication.

Figure 10.23: The rules discovered by RIPPER for the second collection.
As long as the user selects and uses message templates (containing FLC information) when composing messages (replying, forwarding, etc.), the messages will be labelled accordingly by the system and collected into user-relevant threads. Also, the way the prototype is implemented, storing anything in the Tasks folder forces the messages to be classified. That is, the user has to select a thread for every message (or start a new thread)—a message must belong to a thread if stored in the Tasks folder. Regarding this storage and categorisation of information, we can relate it to a dilemma stated by Lansdale (1988): the more time a user has to spend on categorising an item, the less likely it is that the categorisation will be done at all. The automatic thread-matching that we provide the user with can support her in the decision of what category (thread) to put a message in—and alleviate the dilemma.

With regard to the accuracy and folders, according to Segal and Kephart (2000), given that a text classifier can select the appropriate folder 85% of the time, it will almost always be able to give the appropriate folder within its top three guesses. Segal and Kephart (2000) describe the SWIFTFILE system, which presents the top three guesses to the user as three buttons. In this way the user can swiftly file a message into a suggested folder. This can make e-mail filing a less painful task. By presenting such short-cuts for filing a message to one of the three most likely folders it improves the performance without placing any additional burdens on the user. This method could also be applied to the Threads Browser window in our system (cf. Figure 9.7 on p. 185). The cognitive effort required to decide upon an appropriate folder or thread and locate the icon or menu item representing it is substantial enough that many users fall behind in filing their e-mail. Our model alleviates filing by supporting the handling of threads and suggesting an appropriate thread for a message to the user. Since the model is designed to only make suggestions, there is little cost to the user if a suggestion is wrong—the support is transparent so the user can use the e-mail client’s normal interface to file the message.

By including the To:, From:, and Cc: headers in the text matching algorithm, along with the message body, we also managed to include information about the communicating parties in the data to the TF-IDF matching algorithm. This improved the matching results. Using TF-IDF to match messages that fail matches on message IDs and Subject: headers is a simple yet powerful technique. In the second collection the test subject’s case e-mail tool did not put In-reply-to: headers into outgo-
EVALUATION

In our implementation we chose to match on a message-to-message basis. An alternative is to match on a message-to-thread basis by using a "thread centroid" as the target. There could be advantages with the latter alternative, since we are handling threads in the Tasks folder and trying to sort messages into threads. The approach of message-to-message matching that we used is greedy when compared with the message-to-thread approach, i.e., a message is selected even though the thread as a whole may be more dissimilar.

We have, in parallel with the evaluation described here, also used the prototype in daily life. Apart from discovering bugs in our own implementation, we have also discovered bugs in the software used (parsers, lexers, etc.). The thread-matching algorithm is slow in our current implementation. If we are to deploy the prototype, this has to be optimised. Furthermore, the use of FLC information increases the number of attachments sent in e-mail. The attachments are not large, but increase nonetheless the size of the messages.

How much of a human’s work should be automated by a machine? Customising an e-mail filter to accommodate one’s personal notion of “important” messages, for example, requires a detailed description of a fairly subtle concept. The initial customisation may be done willingly by the user, but she may be unwilling to do the subsequent update of the knowledge. We have shown that we can get feasible IF–THEN rules automatically by using an ML algorithm. These rules are easy for the human user to read.
CONCLUSION
AND FUTURE WORK
Chapter 11
Conclusion

In this chapter we summarise the thesis and list the contributions. We also discuss the originality of the work.

11.1 Thesis summary
The aim of the work presented in this thesis is to introduce support for task management in electronic mail (e-mail). E-mail was originally designed for basic asynchronous communication. Nowadays e-mail is also used for information management and task management, with unsatisfactory results to the user. Many of the tasks performed are unstructured, which means that they cannot be described beforehand as in a standard workflow system. There are also many other sources involved in the tasks, which together with the e-mail client constitute the organisational memory (OM). Also, the functionality provided in the e-mail client is not enough for more advanced undertakings. The solution is to restructure the application domain of e-mail. This we do so as to accommodate a support for task management which at the same time is a complement to workflow. The use of the OM is included in the restructuring. We do the restructuring with the user in the centre, and at the same time with the goal of keeping the simplicity of the usage that is so characteristic of e-mail.
The thesis presents support for task management in e-mail in three steps: a conceptual model, a language, and a prototype. These are also the main contributions of the thesis work.

A conceptual model Starting from a user’s point of view we have developed a conceptual model for supporting task management in e-mail. The conceptual model provides us with a user-centered restructuring of the application domain of e-mail with regard to task management. The focus has been on including unstructured tasks into the model. E-mail tasks are modelled as conversations, i.e., exchanges of messages, in the model. The model also describes how a context for a task, in the form of an electronic environment, is included as an essential part of support for the task management in the e-mail system. This means that in addition to the e-mail client (the user interface) itself and the refilled messages, the messages stored in the outbox, and the aliases, among other things, we can also have other programs and sources (the calendar, the address book, and web bookmarks, for example) as part of a task. In the conceptual model, structured conversations are possible, but non-formal communication and (simple) information-sharing are also included.

In the model, incoming e-mail messages are analysed and classified into either the tasks or the non-tasks category. Messages filed into the tasks category can be prioritised according to different criteria. The classification of messages depends on the user and the electronic environment with which the user interacts. The task tracking part of the model (the Conversation support module) includes the functionality for starting, ending, and reminding of tasks, and also the extraction of information that is relevant to a task, such as task status, people involved, and related documents.

A language We have developed a Formal Language for Conversations (FLC), which is used to describe tasks. FLC is based on speech act theory (SAT), which is used as a reference model for the language. The motivation for this is that SAT would otherwise make the conversations too rigid to model unstructured tasks. We also list related research that have shown this problem with SAT. We show how FLC can be used as a container for task-specific information in two specific scenarios: the scheduling-a-meeting scenario and the making-conference-arrangements scenario. FLC can be used to log conversations (tasks) for later analysis.
CONCLUSION

A prototype We describe a prototype implementation based on the conceptual model that we have developed. The implementation was done by extending an existing e-mail client (EXMH) with the proper functionality, after doing an inventory of what system services and functions were needed. We show how the management of tasks is supported in the prototype. We also show how the electronic environment can be included in the task management via the use of agents.

In order to manage the ad hoc nature of e-mail messages we base our approach on the use of message types. The message types are associated with message templates, which give a preliminary classification of the messages provided by the sender. Task-related information is added to the messages structured with FLC and stored using XML. By doing this we provide for the possibility to log the information and learn from it.

The exchanges of messages that represent conversations are stored as threads in e-mail. Because the standard threading function of e-mail clients is unsatisfactory, and to further support conversations, we have enhanced the threading function of our e-mail client. This has been done by the use of standard text matching and IR techniques. The threading algorithm collects and keeps the corresponding messages together and gives the user information about the order, history, and relation of messages.

Another problem with e-mail clients is the management of filtering rules. The rules need to be created and maintained by the user in order to be useful. We introduce the automatic creation and management of these filtering rules in our prototype. We use a rule-based machine learning algorithm (RIPPER) for this purpose. RIPPER receives implicit feedback for classifications from the user in the form of her refilings of messages. Furthermore, because RIPPER is rule-based, the rules can be presented to the user in plain text, which further improves the sense of control felt by the user. By simply looking at a message the user can tell which rule was triggered by the message when it arrived, and why the message was filed into the folder in question.

Both the threading mechanism and the filtering capability of the prototype have been evaluated and found satisfactory.

11.2 Conclusions
We find that our application belongs to a new class of e-mail applications that like KMAIL (Schwartz & Te’eni 2000) “intertwines e-mail with knowledge management.” We have introduced semantic access into a
messaging system in an office environment, in the same vein as Moore and Kimbrough (1994). Our unique contribution is that we have done this in an environment where messaging is inherently ad hoc, namely Internet e-mail-based communication. Furthermore, the sources for knowledge are found in what we call the electronic environment, which is a part of the organisational memory (OM), and which also includes the e-mail client itself. Specifically, our contributions are as follows:

- a conceptual model for supporting task management in e-mail (see Chapter 4)
- a Formal Language for Conversations (FLC) (see Chapter 5)
- a prototype implementation based on an existing e-mail client (see Chapters 7–8)
- an agent-based architecture for accessing and managing the electronic environment (see Chapters 7–8)
- an inventory of system functions and services in e-mail that incorporate task management (see Appendix B)
- an implementation and evaluation of enhanced threading support in an existing e-mail client (see Chapter 10)
- an incorporation and evaluation of a rule-based machine learning algorithm in an existing e-mail client (see Chapter 10)
- a classification of tasks into eight different types (see Table 1.1 on p. 8)
- a review of related, commercially available groupware (see Chapter 2) and also research message-based systems with task management (see Chapter 6).

Included in FLC is the capability to give tasks a name, which implies that threads can be given a name, which is unique for our implementation. Threads can thus be given names that make sense to the individual or the organisation. The prototype also introduces the concept of “intelligent folders,” i.e., folders that monitor threads and suggest a thread to each new message being stored in the folder.

The conceptual model preserves the ease of use and informal communications that characterise e-mail use, although the functionality is increased so as to incorporate task management. Our addition of support for task management coexists nicely with the functions of asynchronous communication and personal archiving that characterise e-mail usage. The user’s personal strategies for organising information and managing time are unchanged. What we do is add support for the user’s strategies. As opposed to a tool, our support system is designed to help the user. That is,
**CONCLUSION**

it does not perform the tasks on behalf of the user. Moreover, the user can stay in control. The support takes into consideration the following aspects:

- how to represent a task
- how to process a task
- how to provide for the reuse of a task.

Firstly, the representation of tasks is accomplished partly by using a Formal Language for Conversations (FLC) based on speech act theory (SAT) to store and convey semantic information about a conversation, and partly by integrating this language with the threading available in e-mail. It is thus possible to look at one message and see how it relates to other messages. Secondly, the processing of tasks is accomplished by the parsing of the FLC structure, which is represented as XML code in the e-mail message. Thirdly and finally, the provision for the reuse of tasks is accomplished by the use of editors to view and modify the information that we create using FLC and XML in conjunction.

Whittaker and Sidner (1996) proposed the use of conversational threading and semantic clustering to reduce the clutter in the inbox. Furthermore, they suggested a function to mark particular inbox items as requiring action, and the ability to program reminders. Our conceptual model for task management in e-mail coincides with all of these findings:

- **threading** is used to collect messages belonging to the same conversation or task
- **semantic clustering** is used to identify stray messages that do not follow the technical structure of Internet e-mail, but still belong to a certain thread in an active task
- **reminders** can be programmed by the user, which will make messages recur in active tasks.

We have hidden the e-mail filtering system from the user and lifted up the threading function as an important support for information management in the e-mail system. Good information management makes task management easier. The concept of a thread is used as a mechanism to sort and organise messages, at the same time representing a task.

A side-effect of us enhancing the management of threads in e-mail is that the thread can be used as a unit for **browsing** and **retrieving** information. Browsing is a central strategy in accessing the different kinds of information in the prototype—it is one of the main strategies for overcoming the information problem (Marchionini 1995, pp. 7–8) and alleviating
the user’s “anomalous state of knowledge” (ASK). The folders can provide the context in conjunction with the threads for successful retrieval of relevant information.

Good indexing and summarisation of document content are needed for effective retrieval. With this in mind, we note that the name of a thread summarises the information in the messages of the thread in a similar way as the Subject: header summarises the text of a single message. By making it possible to rename threads, and to use a personalised task name for tasks represented by the threads, the results of a retrieval can be enhanced.

We find that a rule-based machine learning algorithm such as RIPPER can be incorporated into an existing e-mail client with satisfactory classification results.

Finally, a conclusion from the review of related systems is that, in order to achieve coordination in an e-mail-based environment, using only SAT is too rigid. To achieve coordination in e-mail we need an AI technology such as a machine learning algorithm or agents.
Chapter 12
Future Work

We started this thesis talking about e-mail-based tasks, office work, organisations and organisational memory. We have modelled this “world” and described an implementation of the model in the thesis. We conclude that much interesting research and field-testing remain, now that we have a prototype. In this chapter we discuss possible future work. We look at extensions to our thesis work from three aspects: the conceptual model, the language, and the prototype. We also look at other possible future work, not categorisable under the three aspects. The chapter is concluded with some final remarks about our work.

12.1 The conceptual model

Regarding the conceptual model, we would like to continue examining the formalisation of tasks. Specifically, how can the construction of more sophisticated templates be supported? Can machine learning be used in this context?

We would also like to examine extensions to the model with regard to the archiving and retrieval of e-mail messages. What is required in order to support different classes or types of messages so as to allow them to be managed in different ways? For example, consider the case of legal compliance. A class could be employed in order to store messages in such a way that a court of law will accept a message retrieved from the class as legally admissible. In examining what is required for legal compliance in the model, we would need cooperation with suitable user groups.
The possibility to partially order tasks and define interdependencies on a higher level than single messages has not been examined in the model. This mechanism could be achieved by letting the user define an order between the different conversations representing tasks in the model. Also, we wish to examine the extension of the model with regard to the analysis and reuse of tasks and task management information.

12.2 The language

Very few illocutionary attitudes are defined in FLC and subsequently employed in the prototype. The illocutionary attitude describes what the sender is doing in sending a message, such as asserting, requesting, or denying something. Extending the list of <attitude> in the vocabulary of the language (see Figure 5.4 on p. 94) so as to include such attitudes as order, suggest, accept request, and deny is straightforward, as suggested by Kimbrough and Moore (1997). However, a small number of illocutionary forces that are categorised into a relatively deep tree—where messages are allowed to inherit properties from other messages—would be ideal for a communication system based on SAT, as argued by, among others, Moore (1998).

The possibility to define different roles of a user is missing from FLC, i.e., specifying if a user is acting as an expert in a conversation or maybe just as a curious bystander. Many e-mail users already use so-called signature files to identify themselves when communicating. Information extracted from the signature could be used to characterise the role of a user and still keeping the simplicity of the e-mail interface. The role information could then be used to characterise conversations, as done in CHAOS (see Section 6.2.3 on p. 116). With regard to task names, the use of synonyms would be feasible to add to the language.

12.3 The prototype

12.3.1 Retrieving and inferencing information

With regard to the eight major user requirements for services in a task support system listed by Croft and Lefkowitz (1984), which we quoted in Chapter 1, p. 11, our conceptual model includes them all. However, the prototype does not. The delegation of tasks needs to be addressed in the prototype, i.e., the eighth requirement in the list: Handling multi-user tasks. Related to handling multi-user tasks and delegation of tasks is the
**Future Work**

*support for other languages.* For example, it is very common to forward a message containing a request of an action to another person who is assumed to be better able to handle the task. This can be repeated, thus creating a chain of forwarded messages. The consequence of this is that the final response can come from a person not originally addressed in the request. It is also not uncommon that the response is in a different language than the original request. If the e-mail systems involved do not support FLC, and also the `Subject:` is changed, the response will be difficult to classify into the correct task since traditional IR techniques are not language independent, as discussed in Chapter 10, p. 213.

How to make *inferences* in the prototype remains to be investigated. By introducing SAT into the domain of e-mail and, furthermore, by making use of information available in the electronic environment of a user, we have made it possible to provide for making inferences about goals of the people communicating. Knowledge about the goals in turn makes it possible to support task management even more.

Related to the inferencing is the use of message templates. We have not evaluated the possibility for the user to modify and create *personalised message templates*, which subsequently can provide for more detailed inferencing. Malone and colleagues (1987b, p. 395) employed both user-specific and general templates in INFORMATION LENS. The use of such user-specific templates may improve the perceived usefulness of our system.

The structure and additional information added to messages by FLC can increase the *quality of searches* performed on folders. When a user requests information about an old task or some other old piece of information that she wants to locate, the FLC structure contains information that the user has provided with her own words. Especially the task name, in conjunction with the persons and documents involved, can increase the chances of relevant hits. We would like to test and evaluate this. Lambrix and Shahmehri (2000) describe a query language that allows for querying documents using content information, information about the *logical structure* of the documents as well as information about *properties* of the documents.

The logging of tasks that is done in our system, finally, creates a new source for knowledge included in the electronic environment. This *task log* can also become a key to locating relevant information.
12.3.2 Filtering and routing messages

Currently, we use a “window” of two months for the filtering rules, i.e., we only include the last two months worth of e-mail messages stored in the system when we create the filtering rules. One direction for future research would be to investigate how the size of this window affects the filtering results. Also, it would be interesting to learn something about how the size of this window and the length of a typical task correlate to each other.

Moreover, the first rule is always triggered in the current implementation. That is, the rules are always evaluated in order. Other alternatives would be interesting to consider. Specifically, examining the filtering correctness of individual rules would be interesting to investigate. The rules could then be sorted based on how good they are, so that rules that perform well can rise to the top of the rules list during usage. In other words, we would like to make the rule creation more incremental, in line with the CAP system (Mitchell et al. 1994). Today we just recreate all of the rules at regular intervals, without taking into account that some of the rules may have more longevity. A related question is whether it is feasible to save the filtering rules that were used with a certain task, and store the rules along with the corresponding task for later reuse.

Finally, the filtering rules could take into consideration the following aspects of the messaging environment as well:

- different virtual addresses (e-mail addresses), which often represent a user’s different roles, e.g., the roles that receive family-related messages, project-related messages, and organisation-related messages, respectively
- different real addresses (IP addresses), which represent a user’s different personae or profiles, e.g., the personae that receive office-related messages and home-related messages, respectively
- different states of mind of the user (modes), e.g., the states in which the user is busy, cool, and curious, respectively (see Takkinen 1997).

These aspects would provide for making the filtering (including the prioritisation and presentation) of messages even more personalised.

An area that we have not investigated is to define specific message handling procedures for each message type, and the combination of the five arguments appearing in an FLC message (sender, receiver, illocutionary attitude, predicate, and context). Moore and Kimbrough (1994) have for

67. An extra dimension here would be the roaming user.
their strictly SAT-based language FLBC shown that with a few message types useful distinctions can be made in the communication, and procedures can be defined for handling each message type. Taking this thought further gives us messages that route themselves without human intervention, from system to system. This is another functionality not explored in the prototype. A related question is whether users really want this functionality.

12.3.3 THREADING MESSAGES

In our implementation we chose to match on a message-to-message basis instead of a message-to-thread basis. However, there could be advantages with the latter alternative. This is because we are managing threads in the Tasks folder and trying to sort messages into threads. An evaluation of this approach could be a future step to take.

An interesting experiment would be to remove the human intervention in the thread classification task and instead use a threshold for the TF-IDF weighting in the threading algorithm to define the borders of threads. This automatic classification could be interesting to evaluate. However, we still need root messages to start the classification task with.

In our prototype, the threading algorithm is tightly connected to one specific folder, namely the Tasks folder, as shown in the working example (see Chapter 9). The Tasks folder acts as a to-do list or a list of pending tasks. From one viewpoint the combination of the threading algorithm and the Tasks folder makes the folder act intelligently—any message that is filed into the folder is parsed and inserted into a suitable thread. Here we would like to investigate whether this is a procedure that fits most users’ habits, or if several “intelligent folders” are needed, e.g., one for each task. The user interface and the system’s functions and services need to be evaluated for usability, e.g., as defined by REAL (cf. Löwgren 1993).

Regarding web-based e-mail and links, Phelps and Wilensky (2000) describe a framework for robust hypertext linking of documents on the web that solves the problem with broken hyperlinks. This is a problem congruent with the broken links between messages in an e-mail thread. Phelps and Wilensky propose the use of URLs augmented with a small “signature,” which is computed from the referenced document. This signature can be fed to a search engine and the related document can thus be located, even though the document has been moved. A similar technique could be used for threading e-mail messages.
12.3.4 IMPLEMENTATION ISSUES

The use of a JAVA-based agent infrastructure such as JATLITE provides for extensions of the prototype where mobile agents can be used, i.e., roaming e-mail users can be supported. The agents can follow the user and thus provide a familiar electronic environment and subsequently also accurate filtering. The use of XML, which is a language-independent data-interchange format, can also make other parts of the electronic environment available to a roaming user and for purposes other than filtering. XML-parsing tools and libraries are now available in a number of programming languages (JAVA, TCL, and PERL, for example), so it would be relatively easy to develop tools for other platforms using the model. There are also emerging standards for threading in HTML (Berman, Resnick, & Shlens 1998).

In the prototype the language FLC has been integrated with an individual user’s e-mail client. Another alternative is to implement support for FLC on a server that manages a mailing list. This could be the case when, e.g., a distributed ad hoc group decides to cooperate on a project. Initially an individual user typically defines an alias for the e-mail addresses of the members of the groups in order to facilitate messages to the group. The next step is to start a mailing list and use the mailing list address instead of the alias. How to implement the FLC mechanism on the mailing list server in this scenario would be interesting to investigate.

12.4 Other areas

It would be interesting also to look at interfaces to standard workflow systems and how our task management model could be integrated into those. Since we use templates to structure the messages, a simple step towards integrating the prototype with a workflow system would be to convert the templates into the forms used in a form-based workflow system. W/MC, the Workflow Management Coalition (2001) has launched Wf-XML, an XML-based process management standard, which could be interesting to examine and evaluate. Wf-XML is based on the W/MC Interoperability Interface which can work with HTTP or a number of other transport mechanisms, including e-mail and direct TCP/IP connection.

There are a number of document formats, such as HTML, XML, Portable Document Format (PDF), and Virtual Reality Markup Language (VRML), that specify documents consisting of a root resource and a number of distinct subsidiary resources referenced by URIs (Universal Resource Identifiers, of which URLs are a subset) within that root.
resource. There is a general need to be able to send this type of multi-resource documents in e-mail messages. Palme, Hopmann, and Shelness (1999) describe how MIME can be used to encapsulate aggregate documents. This structure is reminiscent of the thread structure in e-mail, which implies that threads can be distributed “as is” using e-mail. This can be useful when sharing conversations between each other.

12.5 Epilogue

When we began our work, one of our main points was—still from a user’s viewpoint—to avoid creating yet another icon for the user’s desktop. This is one of the key observations made by, among others, Turner and Turner (1997) when introducing new software to a user. There is a need to integrate new software with other tools, especially if they are used in conjunction—users prefer integration with existing tools to standalone innovations.

It is difficult to get users to accept using one system for sending and receiving e-mail, and a separate system for workflow applications. Many users also dislike workflow systems, because they feel constricted by the formalism (Palme 1995, p. 41). Our work is an attempt to develop something that suits the user’s way of working and using applications. Our support for task management in e-mail is optional—the user can continue to use the e-mail system as before if she wants to, and then incrementally add new functionality: templates, FLC support, enhanced threading, and automated filtering rules. We do not impose a procedure of usage on the user. Our system is rather non-intrusive, although the emphasis on the concept of thread as a unit for managing task information has introduced some new functionality and thereby some new but (we believe) simple procedures to the user.

It is finally with great satisfaction that we note that we have come full circle. We started our research designing and evaluating a user interface called CASUAR (which, by the way, is not an acronym) for filtering e-mail (Takkinen 1994). We continued, investigating the principles for human categorisation of information as related to computerised management of information. The result was a system dubbed CAFE (Categorisation Assistant For E-mail) (Takkinen 1997). Now we close the circle with the (unnamed) system in this thesis, which collects all our ideas under a common umbrella: a conceptual model, a user interface, and a machinery for managing structured information in the form of tasks in e-mail.

68. The Busy mode of CAFE (Takkinen 1997) roughly corresponds to the system presented in the thesis.
REFERENCES
Bibliography


I. The date in italics after the URL is the date when the resource was accessed.


Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


Bibliography


APPENDICES
Appendix A

The Making-Conference-Arrangements Scenario

We use two scenarios to present our ideas: the scheduling-a-meeting scenario and the making-conference-arrangements scenario. The former, more simple scenario we use throughout the thesis in Chapter 4 (the model), Chapter 5 (the language), and Chapter 9 (the prototype) in order to exemplify the use of the concepts introduced. In this appendix we describe the more complex scenario of making conference arrangements.

First we give a general description of the making-conference-arrangements scenario, similar to the description of the first scenario in Section 4.3 on p. 56. Then we use the two viewpoints of the user and the task, respectively, to exemplify the use of the conceptual model with regard to the making-conference-arrangements scenario (cf. Section 4.5 on p. 62 and Section 4.6 on p. 68, respectively). Subsequently, we demonstrate the use of the formal language for conversations (FLC) with regard to the making-conference-arrangements scenario, similar to Section 5.5 on p. 100.

A.1 Introduction

The making-conference-arrangements scenario consists of the communication between a researcher, an administrator and three agents. In the scenario the researcher delegates the task of making conference arrangements to the administrator. The administrator divides the task into three subtasks, consisting of conference-related actions requested from three different agents, e.g., a conference organiser, a travel agency, and a hotel. Specific examples of typical subtasks are reserving a place at the conference,
reserving a flight ticket, and booking a hotel. The reservation of a flight
ticket, which typically also includes the booking of the travel to and from
the airport, is delegated to the travel agency. The booking of a hotel room
and reserving a place at the conference are typically done directly with the
hotel and the conference arrangers, respectively.

Seen from the researcher’s viewpoint, the state diagram for the making-
conference-arrangements scenario is similar to the scheduling-a-meeting
scenario (see Figure A.1 below and cf. Figure 4.2 on p. 57). The difference

![State transition diagram for the making-conference-arrangements from the researcher's viewpoint.](image)

lies in the type of messages sent (cf. Table 5.1 on p. 86). The initial mes-
sage created and sent by the researcher to the administrator is a request for
staff action (states b and c in Figure A.1). In the state of awaiting responses
(state d) several intermediate responses are possible, one for each subtask
created by the administrator. The two alternatives labelling the transition
from state d to e (delivery error or timeout) represent error messages from
the e-mail delivery system and the use of reminders, respectively. When
the researcher receives responses from the administrator the researcher
can decide if the conference arrangements have been made successfully or
not (state $e$ or $f$). In state $f$ the researcher is probably not satisfied with the result, e.g., the flight schedule may need adjustments.

In conclusion, we note that two levels of reporting back results occur in the scenario. Firstly, the agents (the travel agency, as well as the conference organiser and the hotel) have to report back to the administrator. Secondly, the administrator has to report back to the researcher.

**A.2 Exemplifying the conceptual model**

**A.2.1 A user's view of the model in the scenario**

We now exemplify the use of the conceptual model from the user’s viewpoint. A graphical overview of the scenario is shown in Figure A.2. To keep the example small we assume that the task only has one subtask, i.e., the administrator (person 1 in the figure) only communicates with one agent (the conference arranger, person 2). The numbers denote the order of the messages sent in the scenario. The researcher (person 0) initiates the task by sending a message (no. 1 in the figure) to the administrator (person 1). Message no. 2 and the dashed arrow represents an automatic response confirming, e.g., the delivery of message no. 1 to person 1. Person 1 then starts the subtask by sending message no. 3 to person 2. After some time person 2 responds with the result in message no. 4a. In our scenario the same message is also sent to person 0 (message no. 4b)—

---

a. We also do not show the reminders in the figure, even though persons 0 and 1 both create reminders. The reminders are not triggered in this scenario, as compared to the scheduling-a-meeting scenario shown in Figure 4.4 on p. 64.
it is sent as a “carbon copy” (Cc:) to person 0. Subsequently person 1 sends one message (no. 5) reporting the result of the subtask back to person 0, possibly with some additional text.

Seen from the initiator’s (person 0’s or the researcher’s) point of view, the communication appears as it is depicted in the state transition diagram in Figure A.3. We use the same representation as presented in Figure 4.6 on p. 66. We disregard any message delivery errors. The circles represent the states and the filled arrows show the transitions between the states. The hollow arrows show documents that are either produced in a state or input to a state. Each state is described in two ways: an action to the left of the state and a log of the conversation to the right of the state.

We only show the states where the task log is changed. Initially, in state \(a\), the log is empty. The initiator (person 0 or the researcher) starts the communication by composing a request for a staff action. She decides to use a template for this since she knows that this will make the system log and track the exchange of messages concerning the topic of the message. The message template looks like an ordinary message on the screen, but has some additional information in the header about the general purpose of the message: that it is a request and that the message type is staff action. This means that the system should be looking for responses to the message, and also that several responses from the same sender are possible, which are intermediate reports of progress in the task. The messages are linked together in a thread and made available as one unit to the user. When the user (the researcher) has composed and sent the message to the administrator, the log contains the request for a staff action (state \(b\) in Figure A.3). The user also programs a reminder to remind herself about the task, shown in state \(c\) in the figure.

At certain points in time the initiator views the conversation, typically triggered by an incoming e-mail message. In the scenario person 1’s software sends an automatic MDN to confirm that the message from the initiator has been received (see state \(d\)). Since person 2 (the conference arranger) also sends her response to person 1 as a carbon copy (Cc:) to the initiator, the response appears in the initiator’s conversation log in state \(e\) in Figure A.3. After a while the response from person 1 is also received, containing a confirmation of the successful registration at the conference.

---

b. Typically, the administrator also confirms to the agent the message received from the agent (person 2).
Since the task is finished from the initiator’s point of view, the log is stopped in state $g$.

The state transition diagram in Figure A.4 on p. 286 shows the communication in the scenario from the administrator’s (person 1’s) point of view. Initially, person 1’s task log contains the message received from person 0, which is a request for a staff action. Person 1 now starts a task of her own by sending a message (also) of type request for staff action to the confer-
Appendix A

Figure A.4: The administrator’s (person 1’s) view of making conference arrangements in the model.

ence arranger (person 2) (state b in the figure). As in person 0’s case above, person 1 also programs a reminder to remind her to check the status of her task after a certain time (done in state c). In state d person 1 is idle with regard to this task. After a while person 2 responds (state d). In state e person 1 has checked the information received from person 2 in state d and subsequently sends a message to person 0. This message contains a confirmation that a place has successfully been reserved at the conference. The confirmation also contains the original message from person 2, i.e.,
person 1 forwards the message from person 2 to person 0, and also adds some more information.\\textsuperscript{c}

\textbf{A.2.2 A TASK'S VIEW OF THE MODEL IN THE SCENARIO (THE RESEARCHER'S TASK)}

We now describe the use of the conceptual model from the viewpoint of the task, \textit{i.e.}, instead of the different states of the user as in the previous section, we focus on the states of the task.

We only show the researcher’s task since it is congruent to the administrator’s task. The task’s view of the model is shown in Figure A.5 on p. 288 (cf. Section 4.6 on p. 68). The circles represent the task states. The filled arrows with accompanying text represent the enabling activities, which are either activated by the initiator (the researcher, here denoted \textit{user}) of the task or the system. The hollow arrows denote input used in the states, if any. The figure also includes the updated context of the task (shown to the left of the states in the figure). We only show the states where the context is changed.

The task is started (state \textit{a}) and the task context is initialised to contain the task’s name and the (initially) two users involved in the task (state \textit{b}). Also, the context contains a link to the current conversation, \textit{i.e.}, the exchange of messages that represents the task to the user. The initial message in the conversation contains the \textit{request for a staff action}, which has been structured with the help of a message template. The task collects information about the newly started conversation from the personal and external sources available in the electronic environment. The address book, the calendar, and the outbox are three examples of these sources, given in state \textit{c} in the figure. The user’s programming of a reminder is represented by state \textit{d}. The task context is updated accordingly with a \textit{Documents} section, containing a pointer to the reminder, and a \textit{Deadline} section, with data extracted from the reminder definition.

States \textit{e} and \textit{f} in Figure A.5 represent the classification of incoming messages. Note that the \textit{Users} information in the context is updated in state \textit{e} when the “carbon copy” message from person 2 arrives. Compared to the scheduling-a-meeting scenario (see Figure 4.8 on p. 70), we see that the \textit{Users} information changes during the task being performed in this scenario.

In state \textit{f} in Figure A.5 the task context is updated with the confirmation message that the user has received from person 1. The task then receives

\\textsuperscript{c} We include this in the example in order to get a more interesting example when we look at the use of the formal language in Section A.3 on p. 289.
the Stop Logging Conversation command from the user (state g). This indicates to the task to make the classification rules for the conversation inactive in the rules database. Also, the conversation is moved to a separate folder (state h in the figure), for possible later reference or reuse.
A.3 Using the formal language

The formal language for conversations (FLC) and the message types presented in Chapter 5 are here used in the context of the making-conference-arrangements scenario (cf. Section 5.5 on p. 100 and the scheduling-a-meeting scenario).

A.3.1 The message types in the scenario

The message types and the order of the messages exchanged in the scenario are shown in Figure A.6. The purpose of the initial message in the scenario is characterised by the SAC (staff action) message type. The initiator (person 0) selects the template for the SAC message type, composes a message and sends it off (message no. 1 in the figure). Person 1 (the administrator) opens, reads, and interprets the message. In the meantime person 1’s system sends an automatic response (an MDN message—see Chapter 3) telling person 0 that the message has been opened by person 0. This is denoted by message no. 2, which is characterised as a statement or STM message type. Person 1 subsequently sends a new SAC message (message no. 3 in the figure) to person 2, who is the conference arranger.\textsuperscript{d}

The response from person 2 contains the confirmation of the conference reservation. Person 2’s system is not using FLC so the message from

\textsuperscript{d} We can also imagine that the message is of type QFI (query for information), if the information is assumed to be relatively easy for the conference arranger to retrieve, and does not require her, for example, to compile new documents.
A PPENDIX A

person 2 is interpreted as a message of type OTH by person 1’s system (message no. 4a). Person 2 also sends a copy of the confirmation to person 0 (message 4b), similarly of type OTH.

Lastly, the STM message from person 1 (message no. 5) is the response from person 1 to person 0, representing the completion of the task started by person 0. The response consists of the OTH message previously received from person 2, forwarded by person 1 after adding some comments to it.

A.3.2 THE LANGUAGE IN THE SCENARIO

We now exemplify the use of the language (see Figure 5.4 on p. 94 and Figure 5.5 on p. 97) using the making-conference-arrangements scenario. We follow the order of the six messages as shown in Figure A.6 on p. 289. Moreover, we assume that person 0 (the initiator) and person 1 (the administrator) are the only ones who use FLC.

(1) With help from the information provided by the initiator (person 0) via the template for requesting a staff action—and based on the SAC message type—the Conversation support module formats the message using FLC as follows:

```plaintext
msg(person(p0), person(p1), request,
    complete(string("Please, register me for the conference")),
    [sendingMachine(obel10),
    timeSent(2001, 11, 22, 18, 8),
    taskName(string("Conference arrangements")),
    pendingTask(yes)],
    msg241) (A.1)
```

This is interpreted as follows: it is a message from person 0 to person 1 with the illocutionary attitude request. The request concerns a staff action complete, which is the predicate of the predicate statement that follows the attitude. The details of the action are specified in clear text using the string argument. The current context contains the ID of the sending machine and the date and time when the message was sent. The task name is also set by the sender (“Conference arrangements”). The pendingTask component is set by each user; here, person 0 has set the value to “yes,” meaning she wants to track the messages belonging to the task. This is usually the default for new tasks. The message is identified by the number 241.
(2) Person 1’s system receives the request for a staff action message from the initiator. Since person 1’s system handles MDNs (see Chapter 3), an automatic acknowledgement of the received message is sent. The MDN message is formatted as a message of type STM using FLC and then sent to person 0:

\[
\text{msg(person}(p1), \text{person}(p0), \text{assert}, \\
\text{opened(message(msg241))}, \\
\text{[sendingMachine(obel3)}, \\
\text{timeSent(2001, 11, 22, 18, 9)}, \\
\text{taskName(string("Conference arrangements")}, \\
\text{pendingTask(yes)}, \\
\text{respondingTo(msg241)}, \\
\text{msg244)] \quad (A.2)}
\]

This message simply asserts to the initiator (person 0) that the previously sent message was received by person 1’s system. The context of the message contains the ID of person 1’s machine, the time the MDN was sent, and the ID of the message that the MDN is responding to. The number 244 is used to identify the current message (the MDN).

(3) Person 1’s system recognises and categorises the message from person 0 as a SAC message. It also recognises that an action is requested—this is determined from the presence of the \text{complete} predicate in message A.1 above. Person 1 sees the contents of this predicate in the \text{string} component and decides to act. She composes a new message to person 2 (the conference arranger) to perform the action (to register person 0 for the conference). We also assume that she wants more information about the travel and hotel alternatives to the conference so that she can subsequently ask several agents (not shown in the scenario elsewhere). The message from person 1 to person 2 then looks as follows:

\[
\text{msg(person}(p1), \text{person}(p2), \text{request}, \\
\text{and([} \\
\text{msg(person}(p1), \text{person}(p2), \text{request,} \\
\text{complete(string("register researcher X for conference")}, \\
\text{[])}}, \\
\text{msg246a)], \\
\text{msg(person}(p1), \text{person}(p2), \text{query,} \\
\text{send(string("Do you have other hotel and travel)}}
\]
This message consists of two illocutionary attitudes (one request and one query) within the same context, grouped together with the and component in the content part of the message. The request specifies that person 1 wants to register the researcher named X for the conference, and that person 2 should complete this request. Furthermore, the query specifies that person 1 wants person 2 to send information about hotel and travel alternatives to the conference. The query part can be seen as a kind of staff action. However, since the information about hotels and travel is assumed to be easily retrievable by person 2, the illocutionary attitude is query. The context part of the two parts is left empty ("[]"), which implies that the context is available elsewhere—here, at the end of the message. The context contains the ID of the sending machine and the time that the message was sent. The task name has been changed by person 1 to “Register for conference.” The current message is identified by the number 246.

(4) Person 2’s system, which does not make use of FLC, receives the message from person 1. Person 2 completes the requested action and also retrieves the queried information. She sends a reply message back to person 1, and also a “carbon copy” to person 0 (cf. Figure A.6).

(5) When person 1’s system receives the message from person 2 it is classified as a message type of Other (OTH), with the illocutionary attitude Unknown. Person 1 composes a new message to be sent to person 0, which subsequently completes the task started by person 0. She decides to forward the information from person 2 to person 0, but adding some clarifying information to person 0. The message looks as follows:

```plaintext
msg(person(p1), person(p0), assert,
    and(
        msg(person(p1), person(p0), assert,
            interesting(string("I have registered you for the conference. This is the information that I

```

292
THE MAKING-CONFERENCE-ARRANGEMENTS SCENARIO

Firstly, the message is a statement where person 1 assures (the illocutionary attitude) that the information contained in it is true, and also interesting to person 0. The message contains two parts. The first part is a personal message from person 1 to person 0 about the contents of the message, and the task that person 1 has performed on behalf of person 0 (“This is the information that I got”). In the second part is the message being forwarded from person 2 to person 0 by person 1. It is declared as a \textit{simpleUtterance}, which denotes a message within this message. The original message from person 2 is referenced in the context of the \textit{simpleUtterance as msg248}. Looking at the context of the whole message we see that the whole message references the original message from person 0, which started the task (see A.1 above). Also, it is made clear that the message has indeed been forwarded by person 1 (the \textit{forwardedBy} predicate).

(6) Person 0, the initiator, receives the message from person 1. The message is interpreted by the initiator’s system according to the FLC information in it and inserted into the current conversation.

c. The original message received from person 2 has been classified as belonging to the task named “Register for conference,” as defined by person 1. Associated with this task are messages defined as belonging to the task named “Conference arrangements” by person 0. When person 1 forwards the message received from person 2 (identified by \textit{msg248}), the connection can be made from the message to the task of person 0.
Appendix B
Services and Functions
in E-Mail

This appendix lists the services and functions available in an e-mail system. The system described corresponds to an implementation of the conceptual model for supporting task management in e-mail presented in Chapter 4. An implementation based on these services and functions is described in Chapters 8–10.

B.1 Introduction
When a new concept is introduced into a domain, such as our notion of tasks in the form of conversations in e-mail, we must relate it to the system services and the system functions that already are available. In this way the use of the concept, when it is realised in the interface, will not be too difficult to understand by the user (Löwgren 1993).

The services and functions can be classified into both abstract and concrete service and functions, respectively. The abstract services are actualised by the abstract functions, while the concrete services are realised by the concrete functions. Note that there need not be a one-to-one correspondence between services and functions.

B.2 The new and old services and functions
The following lists of services and functions in e-mail are extensions to previous work (Takkinen 1994, p. 68) and introduce new services and functions specific to our conceptual model for supporting task management in e-mail. The new entries are printed in italics.
Table B.1 lists the abstract services and functions, while Table B.2 lists the concrete services and functions. We have omitted protocol-specific functions such as “get new message from server” in the IMAP protocol (see Chapter 3).

Table B.1: Abstract services and functions

<table>
<thead>
<tr>
<th>Abstract system service</th>
<th>Abstract system function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure system</td>
<td>define properties of system</td>
</tr>
<tr>
<td>Communicate with others (main service)</td>
<td>receive message</td>
</tr>
<tr>
<td></td>
<td>send message</td>
</tr>
<tr>
<td></td>
<td>automate communication</td>
</tr>
<tr>
<td>Handle task</td>
<td>start task</td>
</tr>
<tr>
<td></td>
<td>track task</td>
</tr>
<tr>
<td></td>
<td>end task</td>
</tr>
</tbody>
</table>

Table B.2: Concrete services and functions

<table>
<thead>
<tr>
<th>Concrete system service</th>
<th>Concrete system function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get new message</td>
<td>get new message from system</td>
</tr>
<tr>
<td>Display message</td>
<td>open message</td>
</tr>
<tr>
<td></td>
<td>display message type</td>
</tr>
<tr>
<td></td>
<td>reformat message according to type</td>
</tr>
<tr>
<td></td>
<td>decrypt message</td>
</tr>
<tr>
<td></td>
<td>unseal message (electronic seal)</td>
</tr>
<tr>
<td></td>
<td>verify message signature (electronic signature)</td>
</tr>
<tr>
<td></td>
<td>close message</td>
</tr>
<tr>
<td></td>
<td>print out message (on printer)</td>
</tr>
<tr>
<td></td>
<td>open folder</td>
</tr>
<tr>
<td></td>
<td>close folder</td>
</tr>
<tr>
<td></td>
<td>display a summary of folder content</td>
</tr>
<tr>
<td></td>
<td>display message status (new, read, unread, etc.)</td>
</tr>
<tr>
<td></td>
<td>display annotation</td>
</tr>
<tr>
<td>Compose message</td>
<td>fill in message header</td>
</tr>
<tr>
<td></td>
<td>write message text</td>
</tr>
<tr>
<td></td>
<td>reply to a received message</td>
</tr>
<tr>
<td></td>
<td>forward message</td>
</tr>
<tr>
<td></td>
<td>add attachment</td>
</tr>
<tr>
<td></td>
<td>insert (whole or part of) messages</td>
</tr>
<tr>
<td></td>
<td>add personal signature</td>
</tr>
<tr>
<td></td>
<td>create template</td>
</tr>
<tr>
<td></td>
<td>select (and use) template</td>
</tr>
<tr>
<td></td>
<td>change (remove, redefine) template</td>
</tr>
<tr>
<td></td>
<td>write annotation</td>
</tr>
</tbody>
</table>
### Services and Functions in E-Mail

**Table B.2: Concrete services and functions (continued)**

<table>
<thead>
<tr>
<th>Concrete system service</th>
<th>Concrete system function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send message</td>
<td>send message</td>
</tr>
<tr>
<td></td>
<td>encrypt message</td>
</tr>
<tr>
<td></td>
<td>seal message (electronic seal)</td>
</tr>
<tr>
<td></td>
<td>sign message (electronic signature)</td>
</tr>
<tr>
<td></td>
<td><strong>confirm delivery</strong></td>
</tr>
<tr>
<td></td>
<td><strong>confirm reading (opening)</strong></td>
</tr>
<tr>
<td></td>
<td>create automation</td>
</tr>
<tr>
<td></td>
<td>change (remove, redefine) automation</td>
</tr>
<tr>
<td></td>
<td>regret (undo) sent message</td>
</tr>
<tr>
<td></td>
<td>move message to folder</td>
</tr>
<tr>
<td></td>
<td>copy message to folder</td>
</tr>
<tr>
<td></td>
<td><strong>move annotation to folder</strong></td>
</tr>
<tr>
<td></td>
<td><strong>copy annotation to folder</strong></td>
</tr>
<tr>
<td></td>
<td>create automation</td>
</tr>
<tr>
<td></td>
<td>change (remove, redefine) automation</td>
</tr>
<tr>
<td></td>
<td>save a copy of sent message (log)</td>
</tr>
<tr>
<td></td>
<td>save message to file</td>
</tr>
<tr>
<td></td>
<td>print message</td>
</tr>
<tr>
<td>Save data in personal database</td>
<td>move message to folder</td>
</tr>
<tr>
<td></td>
<td>copy message to folder</td>
</tr>
<tr>
<td></td>
<td><strong>move annotation to folder</strong></td>
</tr>
<tr>
<td></td>
<td><strong>copy annotation to folder</strong></td>
</tr>
<tr>
<td></td>
<td>create automation</td>
</tr>
<tr>
<td></td>
<td>change (remove, redefine) automation</td>
</tr>
<tr>
<td></td>
<td>save a copy of sent message (log)</td>
</tr>
<tr>
<td></td>
<td>save message to file</td>
</tr>
<tr>
<td>Organise data in personal database</td>
<td>create folder</td>
</tr>
<tr>
<td></td>
<td>move message to folder</td>
</tr>
<tr>
<td></td>
<td>copy message to folder</td>
</tr>
<tr>
<td></td>
<td><strong>move annotation to folder</strong></td>
</tr>
<tr>
<td></td>
<td><strong>copy annotation to folder</strong></td>
</tr>
<tr>
<td></td>
<td>change (remove, redefine) folder</td>
</tr>
<tr>
<td></td>
<td>choose sorting order in folder</td>
</tr>
<tr>
<td>Search for data in personal database</td>
<td>search for whole or part of content</td>
</tr>
<tr>
<td></td>
<td>search on whole or part of structure</td>
</tr>
<tr>
<td></td>
<td>browse a summary of data</td>
</tr>
<tr>
<td></td>
<td>open folder</td>
</tr>
<tr>
<td></td>
<td>close folder</td>
</tr>
<tr>
<td>Sort out data from personal database</td>
<td>remove annotation</td>
</tr>
<tr>
<td></td>
<td>remove message</td>
</tr>
<tr>
<td></td>
<td>remove folder</td>
</tr>
<tr>
<td></td>
<td>create automation</td>
</tr>
<tr>
<td></td>
<td>change (remove, redefine) automation</td>
</tr>
<tr>
<td>Stop log of conversation</td>
<td>stop tracking exchange of messages</td>
</tr>
<tr>
<td></td>
<td>move message(s) to a folder</td>
</tr>
<tr>
<td></td>
<td>end log</td>
</tr>
<tr>
<td></td>
<td>change automation</td>
</tr>
</tbody>
</table>
**Table B.2: Concrete services and functions (continued)**

<table>
<thead>
<tr>
<th>Concrete system service</th>
<th>Concrete system function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start log of conversation</td>
<td>initiate log&lt;br&gt;start tracking exchange of messages&lt;br&gt;display thread of sent and received messages&lt;br&gt;compact thread of sent and received messages&lt;br&gt;move message to conversation folder&lt;br&gt;copy message to conversation folder&lt;br&gt;create automation&lt;br&gt;change automation</td>
</tr>
<tr>
<td>Request conversation information</td>
<td>display information&lt;br&gt;edit information&lt;br&gt;save information&lt;br&gt;display thread of sent and received messages</td>
</tr>
<tr>
<td>Program reminder</td>
<td>copy message to ticker folder&lt;br&gt;define time and date for reminder&lt;br&gt;define simple action for reminder&lt;br&gt;remove message from ticker folder</td>
</tr>
<tr>
<td>Change look and functions of system</td>
<td>define default values&lt;br&gt;use default values&lt;br&gt;choose user level (novice or expert)</td>
</tr>
<tr>
<td>Handle electronic environment</td>
<td>view automation&lt;br&gt;change (remove, redefine) automation&lt;br&gt;define source facilitator&lt;br&gt;start source facilitator&lt;br&gt;stop source facilitator&lt;br&gt;define source&lt;br&gt;register source with facilitator&lt;br&gt;unregister source from facilitator&lt;br&gt;connect source to facilitator&lt;br&gt;disconnect source from facilitator</td>
</tr>
<tr>
<td>Present directory data</td>
<td>display data&lt;br&gt;display a summary of data</td>
</tr>
<tr>
<td>Search for directory data</td>
<td>search for whole or part of data&lt;br&gt;browse an overview of data</td>
</tr>
<tr>
<td>Request help</td>
<td>display help&lt;br&gt;dismiss help</td>
</tr>
<tr>
<td>No</td>
<td>Author</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------</td>
</tr>
</tbody>
</table>


No 297  **Mikael Pettersson:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.


No 439  **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.


No 485  **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.


No 503  **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.


<table>
<thead>
<tr>
<th>No</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>Ling Lin</td>
<td>Management of 1-D Sequence Data - From Discrete to Continuous</td>
<td>1999</td>
<td>91-7219-402-2</td>
</tr>
<tr>
<td>567</td>
<td>Jörgen Lindström</td>
<td>Does Distance matter? On geographical dispersion in organisations</td>
<td>1999</td>
<td>91-7219-439-1</td>
</tr>
<tr>
<td>561</td>
<td>Vanja Josifovski</td>
<td>Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration</td>
<td>1999</td>
<td>91-7219-482-0</td>
</tr>
<tr>
<td>592</td>
<td>Mikael Ericsson</td>
<td>Supporting the Use of Design Knowledge - An Assessment of Commenting Agents</td>
<td>1999</td>
<td>91-7219-532-0</td>
</tr>
<tr>
<td>593</td>
<td>Lars Karlsson</td>
<td>Actions, Interactions and Narratives</td>
<td>1999</td>
<td>91-7219-534-7</td>
</tr>
<tr>
<td>595</td>
<td>Jörgen Hansson</td>
<td>Value-Driven Multi-Class Overload Management in Real-Time Database Systems</td>
<td>1999</td>
<td>91-7219-542-8</td>
</tr>
<tr>
<td>597</td>
<td>Vivian Vimarfund</td>
<td>An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories</td>
<td>1999</td>
<td>91-7219-544-4</td>
</tr>
<tr>
<td>598</td>
<td>Johan Jenvald</td>
<td>Methods and Tools in Computer-Supported Taskforce Training</td>
<td>1999</td>
<td>91-7219-547-9</td>
</tr>
<tr>
<td>599</td>
<td>Magnus Merkel</td>
<td>Understanding and enhancing translation by parallel text processing</td>
<td>1999</td>
<td>91-7219-614-9</td>
</tr>
<tr>
<td>601</td>
<td>Silvia Coradeschi</td>
<td>Anchoring symbols to sensory data</td>
<td>1999</td>
<td>91-7219-623-8</td>
</tr>
<tr>
<td>618</td>
<td>Jimmy Tjäder</td>
<td>Systemimplementation i praktiken - En studie av logiker i fyra projekt</td>
<td>1999</td>
<td>91-7219-657-2</td>
</tr>
</tbody>
</table>

**Linköping Studies in Information Science**

<table>
<thead>
<tr>
<th>No</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Karin Axelsson</td>
<td>Metodisk systemstrukturering - att skapa samstämmighet mellan informationssystemarkitektur och verksamhet</td>
<td>1998</td>
<td>91-7219-296-8</td>
</tr>
<tr>
<td>3</td>
<td>Anders Avdic</td>
<td>Användare och utvecklare - om anveckling med kalkylprogram</td>
<td>1999</td>
<td>91-7219-606-8</td>
</tr>
<tr>
<td>4</td>
<td>Owen Eriksson</td>
<td>Kommunikationskvalitet hos informationssystem och affärsprocesser</td>
<td>2000</td>
<td>91-7219-811-7</td>
</tr>
<tr>
<td>5</td>
<td>Mikael Lind</td>
<td>Från system till process - kriterier för processbestämning vid verksamhetsanalys</td>
<td>2001</td>
<td>91-7373-067-X</td>
</tr>
</tbody>
</table>