Carnegie Mellon University, School of CS, Software Secretary Internal Report, SoSec-103, June 1993

CAP II: Making the Calendar Apprentice an Agent

Siegfried Bocionek*, Anand Chandani , Ratul Puri,
Dayne Freitag , Ryusuke Masuoka†, Tom M. Mitchell

Carnegie Mellon University
School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
{bocionek, anand, ratul, dayne, masuoka, mitchell}@cs.cmu.edu

June 6th, 1993

Abstract

The calendar apprentice CAP II, a learning agent that supports his user in scheduling meetings. CAP II is the successor of CAP, a single-place apprentice that gives advice to its user when she or he enters new data in the online calendar program.

CAP II extends CAP in providing facilities that support the autonomous negotiation of meeting dates via email. That means, once the user has entered the wish for a meeting, a proposal is generated and sent to all desired attendees. If they possess a CAP II system, all agents negotiate until they succeed in finding a commonly accepted date. This feature, *asynchronous and (to some extent) user-independent negotiation*, makes the apprentice an agent.

An important feature of the CAP II system is the possibility to include non-CAP users too. That means, everybody who does not possess a CAP II agent can be included in negotiations as long as she or he is reachable by email. This possibility is provided by the semi-structured email message format and some special, non-CAP user supporting mechanisms.

CAP II agents are finite state machines (FSM) that interact by means of a contract-net like protocol. The agents operate strictly locally in a purely reactive manner without any planning. They use its owner's calendar but do not have access to the calendars of other participants. All synchronization works through communication and negotiation.

This paper discusses the needs and requirements on CAP II, as well as our solution, to make it an autonomous agent. The emphasis is, firstly, put on the negotiation protocol and automaton structure, and secondly on the message format that must be acceptable for agents as well as human addressees.

*Visiting Scientist from Siemens Corporate Research and Development, ZFE ST SN 33, Otto–Hahn–Ring 6, 81739 MUNICH, Germany

†Visiting Scientist from Fujitsu Laboratories Ltd., Kanagawa, Japan

Contents

1 Introduction **5**
- 1.1 Objectives of CAP II . . . 5
- 1.2 Definitions: Apprentices and Agents . . . 6
- 1.3 Apprentices are Programming by Demonstration Systems . . . 7
- 1.4 Related Work . . . 8
- 1.5 Calendar Programs on Workstations and PCs . . . 10
  - 1.5.1 Calendar Programs on Unix Workstations . . . 10
  - 1.5.2 Calendar Programs on the Macintosh . . . 12
  - 1.5.3 Calendar Software on PCs . . . 14

2 Autonomous Negotiation Determines the Protocol **14**
- 2.1 Requirements on Negotiation . . . 14
- 2.2 The Protocol . . . 15
  - 2.2.1 Setup of New Events . . . 17
  - 2.2.2 Moving of Events . . . 18
  - 2.2.3 Cancellation of Events . . . 19
  - 2.2.4 Interactive User Intervention . . . 20
  - 2.2.5 Time-Out Handling . . . 21
- 2.3 The Automaton . . . 27
- 2.4 Discussion . . . 30

3 Including Humans Influences the Message Format **32**
- 3.1 Semi-Structured Messages in CAP II . . . 32
  - 3.1.1 Requirements and Solutions . . . 32
  - 3.1.2 The "Good-will" Assumption . . . 38
- 3.2 Messages and LISP Records in CAP II . . . 39
  - 3.2.1 Setup New Meetings . . . 39
  - 3.2.2 Moving of Meetings . . . 44
  - 3.2.3 Cancellation of Meetings . . . 46
  - 3.2.4 Exchange of Information about Meetings . . . 47
- 3.3 Parsing of Semi-Structured Email Messages in CAP II . . . 48
- 3.4 Discussion . . . 51

4 The Implementation **52**
- 4.1 The Architecture of CAP II . . . 52
- 4.2 The Interface between CAP and CAP II . . . 53
  - 4.2.1 The Communication Interface . . . 53
  - 4.2.2 Mapping CAP II Events to CAP Events (and vice versa) . . . 54
- 4.3 User Interaction with CAP II . . . 55

5 Conclusion **56**

6 Future Work **56**
6.1 Extensions of CAP II and Ongoing Projects 56
6.2 Our Vision: The Software Secretary 61

List of Figures

1 Communication of CAP II agents via email 15
2 One-to-many protocol in CAP II with two-way handshake 17
3 A bid reminder message as sent from the host to an attendee 26
4 A last-reminder bid message as sent from the host to an attendee 27
5 Transitions during an event negotiation on a host's site 28
6 Transitions during an event negotiation on an attendee's site 29
7 A meeting proposal generated by a CAP II agent 33
8 A help message as provided for the sender of an unparsable bid 38
9 Initial proposal sent from the host to each attendee 40
10 A Yes bid sent from an attendee to the host 41
11 A No bid sent from an attendee to the host 42
12 A Not-then bid sent from an attendee to the host 42
13 A Maybe bid sent from an attendee to the host 43
14 A counter-proposal bid sent from an attendee to the host 43
15 A confirmation sent from an host to the attendee 44
16 A validation sent from an attendee to the host 45
17 A move request sent from an host to the attendees 45
18 A move request sent from an attendee to the host 46
19 A no-move sent from an host to the attendee 47
20 An cancelation is sent from an host to the attendee 47
21 A cancel request as can be sent from attendees and the host 48
22 An information indication is sent from the host to attendees or informees 49
23 A information request sent from an attendee to the host 49
24 The process structure of CAP II 53

List of Tables

1 The top-level algorithm of the time-out handler 22
2 An example of a reaction descriptor in an event's reaction list 23
3 The algorithm of function `any-condition-yields-true` 23
4 The algorithm of procedure `perform-time-out-reactions` 24

1 Introduction

The Calendar APprentice (abbreviated as CAP) is an interactive calendar tool that gives its user advice on how to set up meetings. In particular, the suggested data (meeting time, date, location, duration) are presented as defaults in the CAP prompt line when those values are requested during, for instance, an "add-meeting" command.[1] The user can then accept the suggestion, by just striking the return key, or enter another value if he is not satisfied.

Because it uses the examples given by its owner to *learn* his preferences about when and how to set up meetings, CAP's ability to give advice improves over time. The data for learning are not only those associated with meetings (time, date, location, duration), but also records about the participants (affiliation, position etc.) or the events themselves (type of event like seminar, course, meeting etc.). Those records are collected interactively each time the user mentions the name of a person or location that is not yet known by CAP. CAP learns each night by activating some Theo routines [15] that operate on the event data collected during the day. The way CAP learns new rules for advice-giving is described in detail in previous publications [9, 6].

In this report we present the design of a successor of CAP, simply called CAP II. While CAP was a (passive) tool for calendar management, CAP II will be an *active agent* that is able to interact with other CAP II agents (and even humans without CAP II) to negotiate meetings autonomously. The communication is performed by email exchange and later also by EFORMS (Electronic FORMs). CAP II is intended to make the process of scheduling meetings considerably more convenient by saving much of the time usually needed for phone calls, checking and reminding, especially when meetings have to be bounced or re-scheduled. Furthermore, if the agent's owner cannot be reached, the agent can act on his behalf and negotiate tentative meetings until the user returns and can finally confirm.

1.1 Objectives of CAP II

Two major objectives were defined for the development of CAP II:

- CAP II should be able to negotiate meetings autonomously by sending and receiving email and
- CAP II should also be able to involve people that do not possess a CAP agent but can exchange email messages for the purpose of scheduling meetings.

The first objective constrains the ways in which the "autonomy" of CAP II agents can be realized. In particular, it affects the protocol for how they interact, i.e. which reactions are necessary when receiving what type of email (or user command, resp.). We have chosen an approach of communicating, finite automata that behave according to a contract net protocol. This approach is an extension of the work of Sen and Durfee [18] who used such a mechanism for the same task in simulation runs (see 1.4 below). Our particular solution will be discussed in detail in chapter 2.

[1] Advice on date and time does not work satisfying yet.

The second objective has implications for the message formats that can be used for email exchange during a negotiation. If only CAP II agents were involved, almost any structured format would suffice. The only selection criteria would be brievity and ease of parsing. When humans are involved, two much more important criteria have to be taken into account:

- The human receiver should accept the messages presented to him and not be bothered by artificial word sequences which obviously were generated by a computer program.
- Composing an email message which is to be processed by a CAP II agent should be as natural for a human without such an agent and should involve no extra writing. By the same token, the receiving agent must still be able to extract necessary information.

In fact, there is a whole range of possibilities, from character-oriented formats to a pure natural language (NL) interface. The former would not meet the mentioned criteria at all. The latter would do perfectly, but would also make the parsing problem quite difficult. Therefore, a decision must be made which balances these opposites. Our solution provides "semi-structured" messages, i.e. "natural" text as it is usually typed within email for the setup of meetings, but with special keywords (like suggest, confirm, cancel) at certain positions. A few keyword-value pairs at the end of the text contain the most important data of the meeting (date, time, duration, location, attendees, topics). In addition, the text contains a few lines of help for the non-CAP participant which suggest how to respond so that the message can be "understood" by the answer-receiving CAP II agent.[2]

In chapter 3 we will discuss our approach and give examples for every kind of message that can be sent or received during a negotiation. A further possibility, called EFORMS (electronic forms), will also be mentioned in the future work in chapter 6. Such a mechanism would facilitate both the composition of semi-structured messages, since it would involve the simple checking or filling out of input fields on the screen, and the parsing of such messages, inasmuch as it would provide rich contextual information and well-defined "field semantics."

1.2 Definitions: Apprentices and Agents

We call CAP an *apprentice* system because it is an interactive program that gives advice and assists the user (here in the office work domain).

We call CAP a *learning apprentice* system because it watches the user through his normal use of the system and improves its advice over time.[3] The improvement is based on learning the personal preferences of the user by taking every completed calendar action as a training example. After learning, the advice of the agent (i.e. the default values offered to the user) will be correct more often, and the user may not need to overwrite data as many times as before. That means, the agent continously performs its job *more effectively*. This is exactly H. Simon's definition of a learning system [19].

We call CAP II an *agent* system because

[2] Like "You can make your reply computer-understandable by beginning your message with one of the following words: Yes, Not-then, Maybe, No."

[3] Brustoloni [4] uses the term *adaptive agent* instead of apprentice. However, he mentiones only the learning of *planning knowledge* which is not required in CAP II.

- CAP II agents *act asynchronously* (in the background) as an additional process to the calendar program when performing negotiations. By this means the agent can react to all kinds of external input (user commands, incoming email, etc.).
- CAP II has some *decision capability*, i.e. it is allowed to select processing steps for new scheduling tasks without disturbing the user (although it can ask if complications occur).
- CAP II agents have a *long-term existence*, i.e. they can keep all (or much of the) acquired knowledge for future use. This property is the basis for both: for learning, as well as for the *reliable* acting on the user's behalf.

1.3 Apprentices are Programming by Demonstration Systems

As stated above, CAP II is a learning apprentice system because it watches the user through his normal use of the system and improves its advice over time. I.e. the knowledge for the advice generation in CAP II is *not (too much) pre-programmed*, but acquired from the user. The user *demonstrates* it by operating a tool as he is used to.

Therefore, consistent with B. Myers' definition in [16], CAP II provides a *demonstrational user interface* and allows its user to "program" the way how his preferences should be handled by the agent. He does this by just using the calendar tool. As the only demonstrational technique according to B. Myers' enumeration - and in addition to the regular tool use - *questioning and answering* is provided. The result of this kind of programming is "high-level" abstractions (the rules about the user's preferences as found by ID3 and/or backprop learning; [9]). No programming language must be learned and mastered by the user. He may not even be aware that he "programs" something during the use of the calender, if nobody tells him. Each adding, moving or deleting of an event is taken as a training example by the CAP II agent.

In the current implementation of CAP II the way how to deal with each different type of email message is determined by the underlying automaton. However, we believe that even this finite state structure can be learned by watching the email exchange, i.e. which response is typed in by the receiver on which message (see also chapter 6). But we also believe that this cannot be done in a completely automated way. A certain portion of user interaction will be necessary to guide and support the learning. For example, the user can be asked "what is the meaning-carrying word/phrase in your message" to determine the search criterion for a specific email type (like "suggest" is the only important word in a CAP II generated email to determine the type *proposal*; see 3). The answer will act as the precondition of a new branch in the automaton.[4]

We believe that our approach - the combination of interactive question-answer techniques together with learning capabilities - is the only way to overcome the weakness of pure machine learning systems as well as the problem of letting computer end users without (or little) programming skill improve a complex system. The presupposition is what we call an *embedded programming by demonstration system or EPBDS* that, as CAP II does, derives as much as possible and asks if necessary.

[4]Such mechanisms of *dialog-based learning* (or *DBL*), however, are not an issue of this report. They are described in detail in [2].

1.4 Related Work

Currently, there is an increasing number of calendar and meeting scheduler applications reported in the CSCW[5] literature [20]. Goldberg et al. [8] have chosen this example as one application for their concept of *Active Mail*. When an agent receives Active Mail, it pops up a convenient Motif-style form, with which the user easily can check fields or enter values. The decision making for negotiation, however, is not performed by the agent; the user does it by looking at the forms and filling them out as desired. Hence, in our terms, Goldberg's meeting scheduler is an agent without decision capability. In this case, the convenience provided to the user by the Active Mail mechanism compensates for the apparent lack. That mechanism is a kind of EFORM implementation like we have in mind for further versions of CAP II. The problem is, that Active Mail requires agents with such a capability at each participant's site. That conflicts with our objective also to include non-CAP users in meeting negotiations (see above).

A mechanism for EFORMS that does not possess this drawback is reported by Borenstein: *Computational Mail* [3]. Instead of sending plain text, "code" for prompting is included in the email and executed by the receiving agent. This results in a line-oriented text window that prompts the user for input. Borenstein argues that an extended email tool that "understands" a relatively small set of well-defined functions (get-number-from-user, get-text, ask-multiple-choice-question etc.) would suffice. Once this tool had been distributed to all users as a replacement for the current facilities, everyone could write Computational Mail applications, like Borenstein's meeting scheduler quite easily. As with Goldberg's calendar, Borenstein's also does not contain autonomous negotiation. This has to be done by the user when answering the prompted questions.

Time management in a surgical clinic in Vienna is described by Egger and Wagner [7]. The task is to schedule doctors and nurses for surgeries, under the constraints of room and equipment availability. A further constraint is that already scheduled surgeries have to be postponed when emergency cases come in. Also to be considered are the positions of the persons involved: university professors (often seen as "gods in white" in many European countries) do as they wish, while nurses have to be present when they are assigned to work. The approach chosen in [7] models the "operation book" used in the clinic, a central calendar to which all people involved have read access, but to which only a few secretaries have write access. Basically, the secretaries set up operations by talking to the people. Now they are supported by an automatic scheduler, that handles a lot of "known" restrictions and tests compatibility of data provided by waiting lists, resource plans and also personal calendars. Furthermore, communication facilities automatically transmit messages to people's PCs in order to inform them of changes. The critical point - besides the need that everyone has to maintain such a calendar accurately - is the access to the personal calendars. This "makes individual and group practices and decisions accessible to discussion and reflection within an organization" [7] – and can obviously be the reason for quarrels and other "unproductive behavior." In our CAP approach, such problems are completely avoided, because we do not have any such centralized system.

"Real" negotiation between distributed parties is considered in a paper by Sen and Durfee [18]. They give a formal definition for the task of meeting negotiation and analyze different

[5]Computer-Supported Cooperative Work.

strategies mathematically. Their experiments, however, were only performed in a *simulation* environment for *contract net* applications, the paradigm they use for the underlying protocol of communicating, finite automata as the agent mechanism (the same protocol and mechanism - both extended to our needs - were chosen in the CAP II system; see chapter 2). Sen and Durfee tried out different *announcement*, *bidding* and *commitment* strategies. The alternatives for an announcement are: send the best meeting time or send the first, say three, best ones (in CAP II we allow only the first strategy). Bidding alternatives are: answer with "yes" or "no", or send counter proposal(s) back (in CAP II, the yes-no strategy was extended to accept also "maybe" and "deny" bids; the strategy with one counter proposal allowed for each attendee in every cycle is currently implemented). Commitment strategies annalyzed by Sen and Durfee are: block the time for an event as soon as you have accepted it, or wait with blocking until a common agreement from all participants is reached (in CAP II the latter approach was implemented by means of a two-way handshake of confirmation/validation messages that is set up after the inviting agent has received agreements from all participants). Sen and Durfee only considered the setup of meetings, and not - as in CAP II - the cancellation and moving of already scheduled dates. Another difference with our system is that they only explore the negotiation problem but no questions of learning, user interface design or email message exchange. Their experimental results verify predicted formulae of the number of necessary cycles and negotiation times. In short, strategies with many proposals need fewer cycles than simple yes-no strategies, but require more complicated, hence more time-consuming, scheduling.

Another interesting *implementation* of meeting negotiation through contract nets was done by Lux [12]. Based on a general "negotiation language" for all kinds of contract-net applications, calendar agents were specified and hooked to existing calendar tools. Lux used the calendar system included in EMACS as well as Sun's Xcalentool. The greatest advantage of his approach seems to be that everybody can rely on the interface he always has been happy with. However, since there was no field test with real users yet, this hypothesis has not been proved. The greatest problem of the approach is that one needs the sources of the existing tools, or at least a detailed interface specification. This assumption holds true only for – usually quite simple – public domain programs but not for powerful calendar systems (see also section 1.5).

As an applicaion of his theory of *structured conversations*, Woitass implemented the calendar scheduling system TVS [23].[6] Each TVS provides a central scheduler called the *mediator agent* or shortly, just mediator. The mediator is responsible to send invitations to all attendees, to collect the answers, and to find a solution where all attendees can participate. In contrast to CAP II, a mediator does not negotiate with another one (although it could, in principle). It "talks" only to human "agents" who have to fill out displayed forms appearing on their screens. The possible message types are very similar to those of CAP II [22]. However, messages are not sent by email. Instead, Woitass has implemented a special X.400 solution which is not as portable as the email approach. Moreover, the involvement of people without TVS in automated meeting negotiations is impossible.

Even more general than Woitass' model to support structured converations is the *Action-Workflow Loop* as introduced by Medina-Mora, Winograd and Flores [13]. It consists of four

[6] TVS is a German acronym standing for TerminVerhandlungsSystem which means a system that negotiates date and time of appointments.

phases: "proposal", "agreement", "performance" and "satisfaction". They use the terms "customer" and "performer" for what we call host and attendee in CAP II. The step "proposal" is the same as in CAP. "Agreement" is the complete process of negotiation until agreement is reached. "Performance" is the final agreement message of the performer; in CAP II terms, a "yes" bid. The "satisfaction" step is the same as sending "confirmations" from the host to the attendees in CAP II. There is no sending back of "validations" in the ActionWorkflow model. Hence, CAP's "common agreement strategy" for commitments (see above) cannot directly be realized. Instead, changing of commitments during every state is possible, something also provided in CAP II. The authors describe a software system for ActionWorkflows that consists of a workflow language interpreter, workflow processors and agent processors (the former maintain the history of negotiations in a transaction database, the latter process the queue of events). Similarly to CAP II, the agents can mix autonomous behavior and user interaction. The usage of ActionWorkflow for a meeting scheduler is not reported, but should be possible.

A calendar system that is very close to our approach was proposed but not yet field tested by Kozierok and Maes [10]. It is a learning calendar apprentice that acquires its owner's preferences and also has the ability to negotiate with other agents via semi-structured email messages (but can hardly involve humans without an agent). The differences to CAP II lie in the details of learning and negotiation. CAP learns concepts from examples (by ID3 and backpropagation), and generates rules from them. Kozierok and Maes's calendar uses a memory-based approach, i.e. each time a decision is to be made the whole history of previous examples is examined to find "similar" cases that help to determine the solution. The negotiation of their calendar agents is based on the presence of a scheduler function. It works as follows: The host first collects from all other agents a list of free time slots. It then determines the set of time slots where all attendees are free. This list is sent to the attendees' agents which respond with a rating, i.e. an ordering of that set. Then, the host agent's scheduler determines the most convenient time and sends it to all attendees as the final meeting date. There is no further confirmation from any side. Rating and selection of the most convenient time is done by the memory-based learning mechanism. Predefined confidence thresholds determine when the user has to be included and when the agent can make such decisions autonomously. Meetings can be changed or cancelled later; however, it is not described in [10], how the protocol works in these cases.

1.5 Calendar Programs on Workstations and PCs

In this section we are going to review some Public Domain systems and some commercial software tools. First we will describe calendar programs on Unix workstations, then on the Macintosh, and lastly on Windows-based PCs.

1.5.1 Calendar Programs on Unix Workstations

The program *calendar* comes with the most Unix systems. The programs *xcalendar* and *calentool* are Public Domain software.

calendar
This program, which comes with most Unix systems. is a simple and completely character-based system. Actually it is just a shell script. The program works in the following way. The user edits the file named "calendar" on his own home directory. (We will refer to this file as "~/calendar.") Each line has to contain a "month-day" date such as 'Dec. 7,' 'december 7,' or '12/7.' (There are some flexibilities.) The user usually chooses the program "cron" (or something like that) to run the program *calendar* at the specified time in a day. When the program is started, first it produces regular expressions for today and tomorrow using an independent program (that is "/usr/lib/calendar"). Then it uses "sed" to get the lines including the regular expressions from the file "~/calendar." The output is sent to the user by e-mail.

The only possible specification for the recurring event is that of "this day, every month." It has no notions of conflicting events nor time. The file "~/calendar" can have include files. (The program "cpp" is used for preprocessing the file "~/calendar.")

xcalendar
This program is written by Roman J. Budzianowski (MIT Project Athena), and Richard Bingle (Purdue University). It is an X windows program based on Athena widgets.

The basic window displays the calendar of a month. Each date on the calendar is a button and clicking the mouse button on the date pops up a notebook for the date. The user can edit the content and save it. The text for each date is saved in a file in the directory "~/Calendar." These files are stored separately and named after the dates. In the basic window, the date buttons that contain some events[7] are highlighted.

There is no way to specify patterns for the recurring events. The program has no notion of conflicting events nor of time.

calentool
This program is written by Bill Randle (Tektronix, Inc.). It has a Open Look user interface and uses "xview" for the realization.

calentool has one window with two parts in it. The top part is the control panel with buttons in it. The bottom part is the calendar display. The user can switch the calendar display between day's, week's, month's, and year's mode by clicking some button in the control panel. (One can also switch from year's to month's, month's to week's, and week's to day's by clicking the corresponding part in the calendar.)

calentool has two kind of events, *appointments* and *notes*. *Appointments* are associated with a time, while *notes* are not. (*Notes* are associated only with dates.) The year's and month's displays show the corresponding calendars with marks for dates containing events.

The week's and day's displays show the events. A week's display is a combination of day's displays for 5, 6, or 7 days in the week. The day's display has a time slot for each half hour from 8:00am to 5:30pm. (These starting and ending times are configurable, while the length of time slot (30 minute) is not.) There are some slots for notes under time slots.

calentool works in the following way when the user wants to enter an event. He has to click the slot type (*appointment* or *note*) and then type in the contents. For *appointments* of more

[7] That means there is a file corresponding to the date in the directory "~/Calendar."

than 30 minutes, one can modify the duration of the event by dragging down the mouse from the time slot. An arrow will appear to show the duration of the *appointment*. The user can have multiple *appointments* starting from the same time slot. In order to do so, one should just click the slot and type in the other contents. The new contents is displayed on top of the previous one, but a "More" button will appear to toggle between these appointments. If all *appointments* starting at the same time slot display their arrows, then the *appointment* on top has the solid-lined arrows while the others get dotted-line arrows.

Events can possess "properties." One can specify, for example, a recurring pattern of the event by those means. The patterns include "Monday to Friday," "Every Day," "Selected Week (1st, 2nd, 3rd, 4th, 5th, Last, All)," "Every Month," "Every Year," "Repeat at specified interval of days," and "Repeat at specified number of times." One can also specify the way of how to signal the user an upcoming *appointment* by "properties" like "Warn in advance by specified number of days" or "Remind a specified number of minutes before."

An event can be cut, pasted, copied, and deleted. If the event is recurring, there are choices between delete (or move) every occurrence or just a today's occurrence. If the latter is chosen, an undelete command reinstates today's occurrence.

The events are stored in the file "~/.appointments" which may possess include files. Though the program can deal only with one file at a time, the appointment file can be set to another file. The author suggests that it will be helpful to set the appointment file to other people's appointment file in an NFS environment.

calentool can also be used to send the user reminder of appointments by email in the way that the program *calendar* (see above) does. A further feature is its ability to print out daily, weekly, and monthly schedules.

The program *calentool* comes with an extensive set of files containing commemorative dates to be used for include files for ~/.appointment.

1.5.2 Calendar Programs on the Macintosh

Though we do not have first hand experiences from usage in the following products, there seem to be quite interesting concepts and ideas involved in them (as we found in some product descriptions).

Microsoft Schedule+

The commercial product *Schedule+* has quite similar concepts and ideas as our CAP II system as concerning the ability of negotiation.[8] *Schedule+* makes it possible to have simple negotiation procedures between users by MS Mail (but without a handshake protocol in order to ensure more robustness and to provide more flexibility in the commitment strategies; see chapter 2.2.1). Here is their sales pitch.

> Microsoft Schedule+ uses a standard calendar as an entry and display form to show you when people are available. Select the best (or only) time, and Schedule+ will

[8] But not the learning capability, and also not the natural language like email interface for people who *do not* posess the calendar software.

book the meeting on your calendar and notify everyone else via MS Mail. It confirms when each participant accepts and books the meeting. Schedule+ prints calendars in notebook sizes, and has alarms to remind you of scheduled events.

The following steps of the calendar administration are done automatically by the *Schedule+* program:

- Determine a meeting time in the owner's calendar and notify the other attendees automatically with a meeting request message about schedule and topic.
- Enter meeting acceptances into the owner's calendar (of course, he can decline meetings and reply with a message giving his reasons).
- Automatically receive a confirmation when a recipient has booked a meeting.

The above three items describe what our negotiation procedures in CAP II perform *without* the extra handshake mechanism of confirmation and validation messages (see chapter 2.2.1). A confirmation in *Schedule+* is what we call a "Yes" bid in CAP II.

In addition to the negotiation capabilities, *Schedule+* provides the user with resizable printing facilities (to fit the schedule to inside any notebook, as the product description claims). Furthermore, *Schedule+* allows its user to define arbitrary parts (arbitrary to some extent only) of his calendar as private such that the *Schedule+* programs of other people don't have access. This feature, in our opinion, is very important if one wants to run a shared calendar environment. The current CAP II system, in contrast, provides *no way* to let agents access the calendars of other attendees. I.e. CAP II is a completely private approach (and we don't intend to change this that soon).

Another nice feature of *Schedule+* is its capability to create public schedules for meeting rooms (and also other resources). The idea is similar to the room apprentice RAP, another apprentice system we have in mind to investigate the possibility of learning negotiation strategies by analyzing exchanged email messages (see chapter 6). The difference is that in RAP we want to learn the apprentice strategy on the client's side (i.e. the person who needs a meeting room). *Schedule+*, in contrast, behaves somewhat like an independent calendar agent for room reservation instead of a human "room administrator" (i.e. the server in the negotiation task).

Other Calendar Software for the Macintosh

The software *calenDAr* by Psybron Systems, Inc. is realized as DA for the Macintosh. Other programs are *DateBook* by After Hours Software and *Easy Alarms 2.0* by Essential Software. Among these products, *DateBook* seems to have high quality. The programs provide the user with calendars, notes and some alarm facilities. Partly, reminder messages can be sent to attendees via AppleTalk. None of these products contain negotiation or even learning mechanisms.

The programs *Mac Project II* by Claris and *MS Project* by Microsoft are, mainly, for project management. Therefore, they also contain some kind of time management facilities which, however, don't exceed the capabilities of the tools mentioned above.

1.5.3 Calendar Software on PCs

In this section we will only describe a calendar program that is part of the Microsoft Windows environment. Partly, the Macintosh software, as discussed in the previous section, is also available for PCs (or will be soon).

Calendar
This program comes with the Microsoft Windows 3.0 or 3.1 package as an accessory program. It has no notion of duration. One can insert only starting times for appointments or other events.

Calendar displays one window. The user can switch between the day's view and the month's view of the window. There are five kinds of marks for the month's view to indicate special dates.

The day's view shows the schedule of the day. The interval between starting times can be set to 15, 30, and 60 minutes. One can also have special times for the appointments which do not fall at these starting times. For example, one can have an appointment starting from 11:19am by setting a special time at 11:19am. One can cut, copy, and paste an appointment. One can also remove appointments during some range of days.

2 Autonomous Negotiation Determines the Protocol

Autonomous negotiation in CAP II means that the calendar agents *exchange email messages* and *react to their contents* in order to solve a common goal, the agreement upon date, time and duration of a meeting. For that purpose, the original CAP program was extended by two additional components as shown in dashed boxes in fig. 1:

- "email": This module receives CAP email asynchronously, parses it and puts the generated LISP record in a "message queue" (see the module structure in fig. 24). Its second job is to construct email from LISP records and send them as replies to other participants.
- "negotiation": This module dequeues messages (if any are present) and processes them, i.e. decides the next step in the negotiation. It calls the apprentice for advice when constructing new proposals or generating bids. After making a decision, "negotiation" transmits a LISP record to "email" with the reply.

The other modules in fig. 1 work as in the first, non-negotiating CAP version [9, 6].

2.1 Requirements on Negotiation

The first objective of CAP II, namely to provide autonomous negotiation between calendar apprentices (and human email users), led to the following, more specific requirements:

1. **Appropriate model of meeting negotiation:** Negotiation between CAP II agents should be as close as possible to the style of human negotiation.

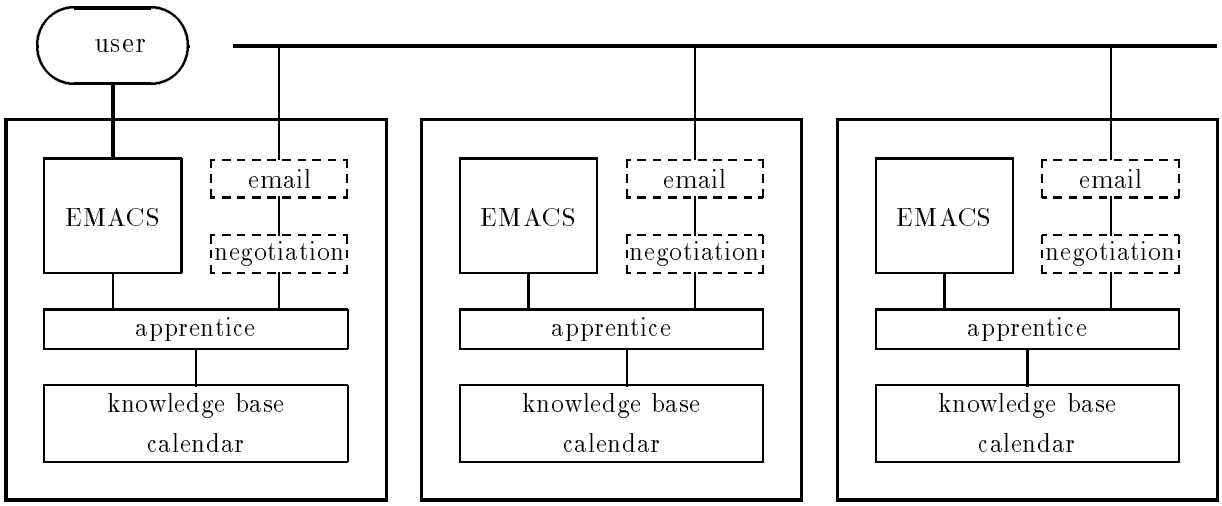


Figure 1: Communication of CAP II agents via email. Each CAP II agent contains the functionality of CAP (solid line modules) as well as additional email and negotiation modules (dashed).

2. **Complete set of services:** CAP II should provide all services to schedule meetings that humans usually use (i.e., invitation, cancellation, change, etc.).

3. **Weak commitment:** CAP II should (also) provide some weak kind of confirming participation on a meeting. That would allow the agents to overwrite time slots with more important events that have been blocked earlier.

4. **Little disturbance:** Autonomous negotiation should disturb the CAP II user as little as possible. The agent should only interact with the user if really necessary.

5. **Immediate reset:** The user of a CAP II agent must be able to reset or overrule any of its decisions at any time.

6. **Exception handling:** A time-out mechanism must be provided to react to missing responses in order that negotiation can proceed.

In the next sections we will show how – and how far – our CAP II implementation meets these requirements.

2.2 The Protocol

As for all distributed systems that communicate through messages, one has to provide a *protocol*, i.e. in our case the rules of sending email back and forth, as well as the semantics of the contents of that email. We followed the suggestions of Sen and Durfee [18] and defined a *one-to-many*

protocol to set up each meeting, although email connections are *many-to-many* by their nature. *One-to-many* means that exactly one party, the initiator or *host*, is responsible for organizing the meeting. The *attendees* talk only to him about that particular event, but not to each other.

There is a third sort of party involved in the setup of meetings, which we call an *informee*. Informees are people who will be notified about events, but do not take part in the negotiation. Their agents receive email messages after a meeting has finally settled or when changes occured and present those messages directly to their users. One nice property of the informee concept is that attendees who cannot decide whether to come or not (we call them "maybe-attendees") can get that status. Thus, they are not involved in the negotiation, but are aware of all information concerning the event. If they finally have time, they can just come to the meeting.

In our design of the roles host, attendees and informees, we gave the host the greatest responsibility – which models fairly well the way people set up meetings, whether by phone or by email. (There are exceptions, especially if attendees are located nearby, so that they can talk to each other door-to-door.) What's more, many people, more than happy to let the responsibility of organizing a meeting rest with someone else, gladly cede control to the organizer. In the final analysis, our one-to-many solution, while it sacrifices some flexibility, represents no real drawback, since we provide *move* and *cancel* services. A participant's agent can later request changes to unrelated events involving some of the same attendees, if the participant prefers some particular date which is already booked. It is not necessary to communicate with these parties during the setup of the new event.

However, there is - so far - no *transitive* movement of events. I.e. if the agent wants to put an event at a certain position in the calendar, it cannot start a *move-request* for another meeting that is already agreed upon and that intersects with the desired time for the currently negotiated meeting. For such a job a *scheduler* would be required that is not yet part of a CAP II agent.

There are also some technical reasons for our decision to provide a host-controlled one-to-many communication. First, the communication overhead in one-to-many systems is usually lower than in systems involving many-to-many message exchange. Second, the greatest computation time for each event is required of the host agent in our model, while the others more or less just generate responses. The same is true with the number of user interactions. Potentially, the agents could defer any decision to their owners, especially if too few examples have been accumulated for real learning. In our opinion, there would be too much interference, if every agent had the possibility to take part in the *synchronization* of each event. A last, technical reason that supports our decision is, that the implementation of the negotiation automaton is easier, because the number of necessary states – compared to the number of participants – grows more slowly than in the many-to-many case.

CAP II agents can handle what we call a *complete* set of negotiation services. I.e., they provide: Invitations, agreement, refusal, cancellation of participation or a complete event, move requests, information requests and information sending, reminders, etc. The next sections will describe all these sorts of communication in detail.

2.2.1 Setup of New Events

Fig. 2 shows our one-to-many communication for the negotiation of a new meeting. The host sends a proposal to all attendees, i.e. date, time, duration, location, attendees and topics of a new event. Each attendee sends back a bid like[9]

- "Yes": I agree.
- "Not-then": I cannot come at that date.
- "Maybe": I cannot decide yet whether I can come.
- "No": I decline to come (for whatever reason).
- "How-about new date and/or new time": I cannot come at the suggested time but I propose another one.

The first four bid values allow us to model what Sen and Durfee call a *yes-no bidding strategy* [18]. This strategy is fully implemented. The fifth bid makes it possible for attendees to send back *counter proposals*, i.e. alternatives for meeting date and time (not yet completely implemented) in an *alternative bidding strategy*. We will discuss only the protocol for the first strategy in the following paragraphs.

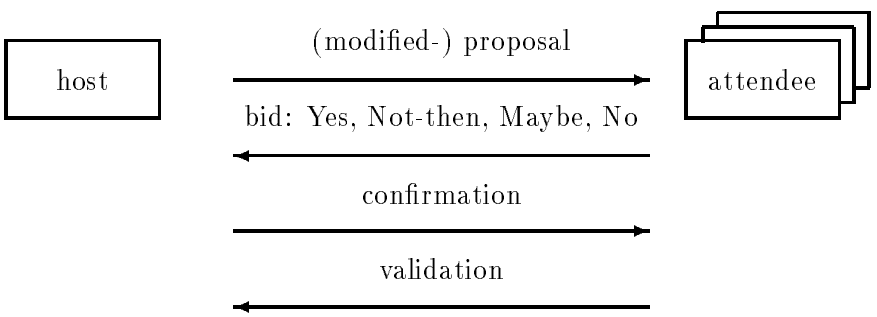


Figure 2: One-to-many protocol in CAP II with two-way handshake

If any of the attendees sends a "Not-then", the host generates a modified proposal and sends it forth to all again. This process cycles until all attendees (exept those who declined) answer with "Yes". The host then sends a "confirmation" message to the attendees. Medina, Winograd and Flores call this step "satisfaction" and point out that it is part of *every* negotiation task in organizations [13]. If this mechanism is used to keep the time of a meeting *tentative* - meaning

[9] The exact syntax can be found in the figures of chapter 3.2.

it can be bounced in order to agree upon other events - until a confirmation takes place, it is the *common-agreement commitment strategy* as described in [18].[10] The attendees finalize the event in their calendars at that point, unless the time slot has been occupied by another event in the meantime. Humans usually would stop meeting negotiation after having received the confirmation of the inviting party.

In extension to [13], who try to model exactly the human behavior during office tasks, the attendees within CAP II negotiations also have to send back a "validation" to the host. Only if all of them send a positive validation does the host fix the event in his calendar. Otherwise, he generates a modified proposal and the negotiation starts again. Therefore, the confirmation and validation steps in our protocol provide a *two-way handshake* mechanism for finalizing events once they have been negotiated. The mechanism is intended to make the results of each negotiation more robust.

One problem with this kind of protocol is the *time-out handling*. The host cannot wait for an eternity for bids or validations to come back. Reasons may be the absence of an attendee, a server problem that prevents email from delivery in time, or a failure of the agent's machine. Nevertheless, the meeting shall take place and the host's agent is "responsible" for that. The solution taken in CAP II is described in section 2.2.5 below.

If attendees send a "Maybe" bid they are put on the list of *informees*, unless the attendee is the only intended participant. Then the negotiation stops and the event is cancelled.[11] Being on the informee list, the "maybe-attendee" will get notified about all confirmations or changes in an event, so that he has every opportunity to attend the meeting later or not (see above).

The fourth bid accepted in the current CAP II version, namely "No", indicates that the attendee declines to come. The reaction of the host's agent is to delete the attendee from the list of attendees. The reaction of the attendee's agent after sending is to skip the event in its list of negotiated events. If the attendee was the only one, the host also cancels the event.

2.2.2 Moving of Events

The host, as well as each attendee, has the right to suggest the movement of an event. However, the semantics are different depending on who has taken the initiative for rescheduling.

If the host has sent an email message of type "host-move-request", together with a new date and/or time and/or duration, the attendees *have to accept* this desire. What they can influence, of course, are the proposed data. Hence, move-requests from the host work exactly like sending a modified proposal during a negotiation cycle. All confirmation and validation status bits are reset and negotiation on that event starts again. This solution models human behavior in which the initiator of a meeting normally makes the decision about whether and how to move meetings, if necessary.

The other possibility is that one of the attendees asks for rescheduling by sending an "attendee-move-request" to the host, meaning that *he can no longer attend at the scheduled date/time*. In this case, the host alone decides whether or not to do it. In the current version

[10] This strategy fulfills the third requirement in 2.1, weak commitment.

[11] Of course, other semantics of the "Maybe" bid could also be implemented, for example, waiting for one more day for a new bid, etc.

of CAP II with only the yes-no bidding strategy, the host, if he agrees to the movement, has to generate a modified proposal and send it to all attendees as a "host-move-request". After that, negotiation for that event starts as described in 2.2.1.

If the host does not agree to the move request, it sends information back to the requesting attendee to tell him that the event is not going to be moved. The attendee's agent (in the current implementation) only presents the information to its user. It is then up to the user whether he wants to live with the decision or cancel his participation in the event (see below).

The decision of the host is driven by some rules and, if necessary, by asking the owner how to deal with the move request. For example, one rule says, if the requesting attendee is the only one, accept a movement (not accepting makes no sense, since the attendee cannot come at the agreed date). In the future, learned features of how the user decided in such cases shall also be taken into account.

2.2.3 Cancellation of Events

There are three possibilities in which cancellation requests for a meeting that is already set up may be sent:

- The host sends a "host-cancellation" to all attendees. This means the event is cancelled, there is no more dicussion about it. No more messages are exchanged.
- An attendee sends an "attendee-cancel-request" to the host, meaning *I cancel my participation in the event.*
- The host sends a "host-cancel-request" to all attendees, meaning *I cancel my participation in the event, but the event shall, nevertheless, take place as it was set up.*

The first case is clear. An initiator of a meeting also has the right to cancel it. The current implementation does not provide for reply messages to ensure that the attendees are aware of the cancellation. Therefore, it may happen that some appear at an already cancelled meeting (if servers are down etc.). An additional "cancel-confirmation", together with a time-out mechanism (see 2.2.5), could prevent such mishaps. We have not (yet) included that prevention mechanism, because, first, we estimate the technical probability for this case is quite low. Second, if the user is absent and cannot (or does not) take a look at his electronic calendar, no further reminder messages via email would help anyway. The only remaining chance, having the confirmation mechanism, is that the host recognizes a problem and tries to contact the non-responding attendee by phone or other means.

In the second case, when an attendee sends an "attendee-cancel-request", (meaning the attendee cancels his participation in the event) the host's agent operates as with move requests. First, it determines whether it should cancel the event completely. This will always be the case, of course, if the attendee was the only one. Other reasons are the *importance* of the cancelling person (expressed by a "status" or "priority" feature in the person's descriptor): If the main topic of the meeting is funding by an industrial partner, and the cancelling person is the company's CMU liaison, then it makes little sense to insist on the meeting without that

person. The decision whether to cancel or not is determined partly by such rules and partly by asking the user (as with move requests; see 2.2.2).

If the host sends a "host-cancel-request" (meaning that he cannot attend, but the meeting should be held as planned), the semantics are as follows: First, the host retains the responsibility to coordinate the meeting, so move or cancel requests can be sent to him later on. Second, each attendee's agent determines whether his owner is still interested in participating in a meeting without the host (this is done partly by rules and partly by interaction with the user). If he is, the event is kept; if not, the attendee's agent itself sends an "attendee-cancel-request" back to the host (see the last paragraph).

The adequacy and the implications of our protocol for setting up, moving and cancelling events are discussed below in 2.4.

2.2.4 Interactive User Intervention

In addition to the exchange of email there must be a mechanism to include the user in the procedure of meeting negotiation, but, according to requirement 4 in section 2.1, as little as possible. Two directions of intercation have to be provided:

- "Agent-to-user": This direction is used if the agent needs support for decisions or wants to deliver information.
- "User-to-agent": This direction is necessary so that the user can become active in order to *influence* an event, e.g. to initiate a new one or to move or cancel a previous one.

The first direction was discussed sufficiently in the three sections above, so we want to outline only the second one here.

In general, the commands of the first CAP version are kept as they were before [9]. They include "add meeting", "move meeting", "cancel meeting." At the end of each function that implements the commands some statements were added that converted the command parameters and Theo data into a LISP structure needed by the CAP II agent (see the examples in 3.2). Then, the structure is put in the shared message queue (see 4.1), as any structure generated by the email parser would be. After that, the user request is handled by the agent like each other request coming from the network. To distinguish user requests from attendees messages, we chose slightly different keywords for the message type (like "usr-proposal" instead of "proposal"). This is necessary for minor technical reasons only (such as setting the *role* value in an event descriptor correctly).

A peculiarity is that interactive user requests are inserted at the very beginning of the message queue, and not at the end like all other requests. Two reasons exist, one technical and one practical. The technical reason is that the user has to wait for the end of an add-event command until he can proceed working with the calendar. If his request were put at the end of the queue, the waiting time might become bothersome in the case of a long queue. In our solution, the user has to wait only for the end of the one request currently being processed. The practical reason for our decision is that, especially for initiating new events, we want to put the information into the network quickly in order to support meeting scheduling at any site as

soon as possible. Experiments so far (see chapter 4) have shown no noticeable slowdown of user interactions with CAP II which we see as a confirmation of our solution.

In general, user interaction (for CAP II owners) is required only when initiating a meeting (i.e., providing the "start data") or requesting a change. As long as no problems occur – like, the agent finds no free time slot within a certain horizon – no additional attention is necessary. When, eventually, the agents have agreed upon a meeting, the user will see it in his calendar. If she or he doesn't like it the cancel and move services can be used to request a change.[12]

2.2.5 Time-Out Handling

Time-out handling means that the host cannot wait forever until bids or validations come back. *Technical reasons* for a missing response may be a shutdown of an attendee's email machine, or problems with servers or disks. Another, more likely reason is just the absence of an attendee, if he is a non-CAP user or, as a CAP II user, had not had the opportunity to answer a question his agent is waiting for. Nevertheless, the meeting shall take place and, in our *host-oriented responsibility model*, the host's agent has to take care for that (this decision was motivated mainly by the way human beings behave; see also section 2.2).

For CAP II's time-out handling we have chosen an approach similar to the way exception handlers in process-oriented computer languages work, e.g. in ADA. The basis is an additional sub-process "time-out handler".[13] This process becomes active all 30 seconds (that interval can be configured) and checks the so-called "time-out reaction list" for each event under negotiation. Roughly speaking, the time-out reaction list consists of a list of *conditions* and a list of *reactions*. Furthermore, a *next-time* value indicates *when* the next reaction(s) should be performed.
Table 1 shows the top-level algorithm of our time-out handler. The process loops until a global flag `*global-stop-signal*` is set to `true`. In that loop the following major steps are performed every 30 seconds:

- Lock the `*global-events-list*`.
- Determine the current time `"now"`.
- Delete all events from `*global-events-list*` that have passed by now.
- For each event under negotiation, and where the processing agent is the host, take the list of attendees and the time-out reaction list, and perform all necessary reactions. Update the reaction times of all reactions that might become necessary next.
- Unlock the `*global-events-list*`.

Note that the value `rd.next-time` is compared to the current time `now` in order to find out whether any reaction must take place at this moment at all. If yes, the conditions are tested

[12] Note, that this totally autonomous behavior does not fully work yet since advice on date an time in CAP II is not good enough so far. Instead, the user is asked (sometimes) whether to accept a certain date etc.

[13] The process structure of CAP II is described in section 4.1.

```
PROCESS time-out-handling ()
  WHILE NOT *global-stop-signal* DO
    lock (*global-events-list*)
    now ← get-time ()
    delete-passed-events (*global-events-list*, now)
    FOR EACH event e IN *global-events-list*
    WHERE e.role = "host" DO
        FOR EACH attendee a IN e.attendees DO
          FOR EACH reaction-descriptor rd IN e.time-out-reaction-list DO
            IF rd.next-time <= now
            THEN
              IF any-condition-yields-true (rd, e, a)
              THEN perform-time-out-reactions (rd, e, a, now)
            update-reaction-times (rd, e, now)
    unlock (*global-events-list*)
    cond-sleep (30)
  END-WHILE
END-PROCESS time-out-handling
```

Table 1: The top-level algorithm of the time-out handler

(call of function `any-condition-yields-true`; see below). If any of them yield true, the list of reactions is processed (call of function `perform-time-out-reactions`; see below).

Each reaction of the "time-out reaction list" is structured as shown in table 2. The `type`, e.g. `missing-bid-reaction` as in the table, is used to branch in some of the time-out handler's functions. `next-time` defines a specific time *when* that particular reaction shall be tested and, perhaps, be processed next. `conditions` is a list of predefined, boolean functions with one event and one attendee as parameters (like `bid-unknown` in table 2). `reactions` is a list of 5-tuples *<procedure, next-time, next-interval, last-time, done-tag>*. The *procedure* is called with the reaction descriptor, one event, one attendee and the current time `now` as parameters. *Next-time* is an absolute time specification that defines when each reaction shall be called the next time. *Next-interval* is a time interval, e.g. 24 stands for 24 hours (if a specific global scaling variable is set to 3600). It is added to the current time after a `send-first-reminder` or an earlier `send-next-reminder` has been performed (as long as the condition holds true and *last-time* has not been reached). *Last-time* is an absolute time specification that defines when each reaction shall be called the last time. Not all time-slots have to be set in all reactions (see below for a list of possible conditions and reactions, and for the "semantics" of the time-slots). The *done-tag* indicates which of the actions are `done`, which ones are `active`, i.e. have perhaps to be performed next, and which ones are `waiting`, i.e. might be performed in the future.

The only condition in table 2 is `bid-unknown`. As stated above, it is a boolean function called - with an event and an attendee as parameters - by the function `any-condition-yields-true` (see table 1). Table 3 shows the algorithm of this function. The interpreting function `eval-cond`

REACTION-DESCRIPTOR
type
missing-bid-reaction
next-time
12-24-92-20-00
conditions
bid-unknown
reactions
send-first-reminder, 12-22-92-20-00, nil, nil, active
send-next-reminder, nil, 24, 12-29-92-20-00, waiting
send-last-reminder, 12-30-92-08-00, nil, nil, waiting
assume-yes-bid, 12-30-92-08-00, nil, nil, waiting
notify-user, 12-30-92-08-00, nil, nil, waiting
delete-attendee, nil, nil, nil, nil, nil
make-informee, 12-30-92-08-00, nil, nil, waiting
END-REACTION-DESCRIPTOR

Table 2: An example of a reaction descriptor in an event's reaction list

is used to evaluate all kinds of conditions. In the case of a `bid-unknown` condition, for example, the expression `event.bids[attendee] = "unknown"` has to be calculated.

FUNCTION any-condition-yields-true (reaction-desc, event, attendee)
result ← nil
FOR EACH c **IN** reaction-desc.conditions **DO**
result ← result **OR** eval-cond (c, event, attendee)
IF result **THEN BREAK**
RETURN result
END-FUNCTION

Table 3: The algorithm of function `any-condition-yields-true`

The reactions in the reaction descriptor of table 2 contain functions for the automatic reminding of an attendee who did not send a bid (like `send-first-reminder` etc.), for the notification of the agent's owner (like `notify-host`), and for automatic status changes of the attendee (like `make-informee`, where the attendee gets the role of an informee only). The time-slots of each reaction can define a specific time (like `12-22-92-20-00`, i.e. Dec, the 22nd 1992 at 20:00, in `send-first-reminder`), but also a time interval (like `24` in `send-next-reminder`; see above). A time defines the date and the hour when to perform the action next or the last time. A time interval, like the 24 hours in function `send-next-reminder` of table 2, is added to the current time `now` when the function `perform-time-out-reactions (r, e, a, now)` determines the new value for `next-time` (see below).

The *done-tag* of each reaction is used to mark the status of each reaction, i.e. has it to become `active`, is it `waiting`, or has it be `done`. Time-slots and *done-tag* are tested by calling the boolean function `perform-it` inside the procedure `perform-time-out-reactions` (see below). Table 4 shows the algorithm of the procedure `perform-time-out-reactions`. The procedure `eval-reaction (r, event, attendee, now)` works as follows:[14]

- Test whether `r.next-time`, compared to `now`, and `r.done-tag` require the evaluation of this reaction.
- If yes, evaluate it; in the case of `send-first-reminder`, for example, an email is constructed and forwarded to the attendee who had not sent a bid yet.

PROCEDURE perform-time-out-reactions (reaction-desc, event, attendee, now)
  FOR EACH r **IN** reaction-desc.reactions **DO**
    IF perform-it (r, r.next-time, r.done-tag, now)
    THEN eval-reaction (r, reaction-desc, event, attendee, now)
END-PROCEDURE

Table 4: The algorithm of procedure `perform-time-out-reactions`

For the moment, we provide a fixed set of functions that can be used as conditions or reactions, respectively (see also table 2). We have yet implemented the conditions

- `bid-unknown`: This function tests whether the bid of a particular attendee still has the value `"unknown"`.
- `validation-unknown`: This function tests whether the validation of a particular attendee still has the value `"unknown"`.

and the reactions

- `send-first-reminder`: The *next-time* of this reaction defines *when* to send an email message of type `reminder` (see fig. 3 below) the first time. This reaction can happen only one times for each attendee.
- `send-next-reminder`: The *time-interval* of this reaction (as a default, expressed in hours) is added to the current time after having performed a `send-first-reminder` or an earlier `send-next-reminder`. The added times will be put in the slot *next-time*. A `reminder` email message will be sent to an attendee each time *next-time* and the `done-tag` require it, but not longer as specified in *last-time*.

[14] Note, that the parameters of all procedures are somewhat redundant for sake of efficiency. The complete events and descriptors are passed to them to *allow* the programming of side effects in the reactions (although there are – nearly – none in the default reaction procedures).

- `send-last-reminder`: The *next-time* of this reaction defines *when* to send an email message of type `last-reminder` (see fig. 4 below). This reaction can happen only one times for each attendee.

- `assume-yes-bid`: Choosing this reaction, the bid of the attendee is set to `Yes` if the time, as defined by *next-time*, is reached. After that, the agent can start sending confirmations, for example, if this particular attendee was the last one who's bid had been missing. An analogous reaction is `assume-yes-validation` if the host's agent is waiting for an attendee's validation.

- `notify-user`: The *next-time* of this reaction defines *when* the agent has to present a message at his owner's screen in order to indicate that something went wrong with contacting an attendee.

- `delete-attendee`: The *next-time* of this reaction defines *when* an attendee, who continously does not respond, shall be deleted from the list of attendees (if the user wants that).

- `make-informee`: The *next-time* of this reaction defines *when* an attendee, who continously does not respond, shall be put on the list of informees. From then on, he will be informed about changes etc., but will no longer take part in the negotiation.

The above set of functions can easily be extended if necessary. Note that the host does not need to select the sending of reminder messages at all. By using reactions like `delete-attendee` or `make-informee`, he has all possibilities to let the negotiation process continue even if bids or validations are missing.

From the functions enumerated above, the reaction descriptors are built when the host initiates a new meeting. There is a little dialog where the host is asked for the conditions, the reactions, and the time-slots. By now, only dialogs for the reaction types `missing-bid-reaction` and `missing-validation-reaction` are provided. As a default setting, `send-first-reminder` will happen 24 hours after the first invitation (or confirmation), `send-next-reminder` will take place every 24 hours from then, and `send-last-reminder` will be activated 24 hours before the intended meeting time. For the same time as `send-last-reminder`, the *next-time* slots of `notify-user` and `make-informee` are set. Both sets of reactions are defined analogously for the reaction types `missing-bid-reaction` as well as for `missing-validation-reaction`.

Note that the function `perform-time-out-reactions` makes also use of additional information about attendees, like their "importance" or other classifying properties. This means that sending a reminder does not happen in any case, but may be restricted by some constraints like "don't send more than one reminder to a company's director" or so. The dialogs to acquire such information become shorter over time as the agent "learns" how to deal with particular attendees (see [2] for a discussion of dialog-based learning mechanisms included in CAP II).
It still has to be discussed how the agents on the host's and attendee's sides react in the case of reminders. This means for the host's agent, what has to be done if the time-out handler processes a `send-first-reminder` or `send-next-reminder` reaction? The solution, chosen for CAP II, works as follows. The time-out handler constructs a LISP-record of type `reminder` or

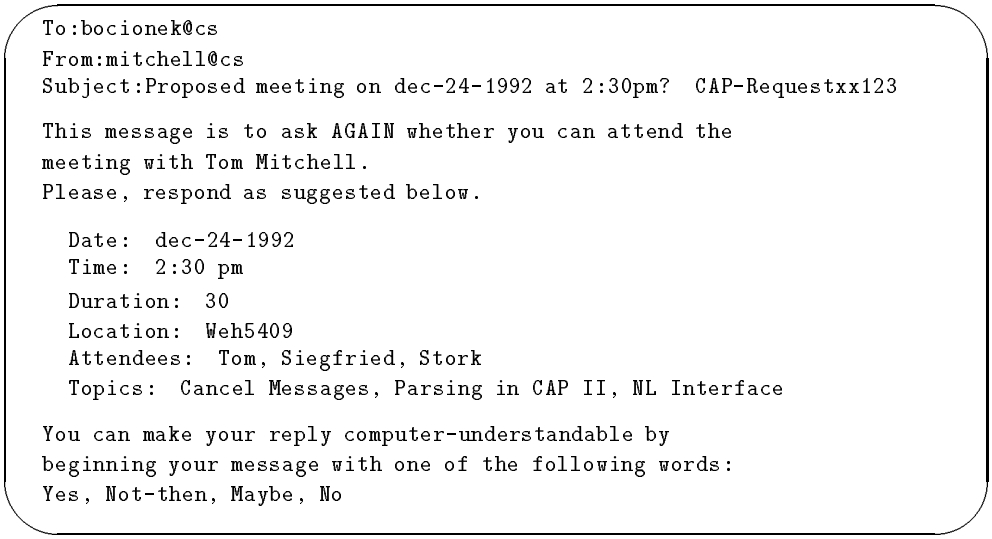


Figure 3: A bid reminder message as sent from the host to an attendee

`last-reminder`. This record is put in the *shared global message queue* and processed by the automaton like any other interactively given user request (see sections 2.3 and 4.1). From that record the agent generates the reminder message shown in fig. 3 and sends it to the attendee that is specified in the record. In the case of a validation reminder the key-phrase is ...`whether you can validate` ... instead of ...`whether you can attend` ... here.

If the attendee's agent receives such a message, it first checks whether it had already sent a bid. If yes, it deletes the message. If no, it branches according to the type of the message. If the type was `reminder`, it generates the bid and sends it to the host. If it was `last-reminder` sending a bid makes no sense because the attendee has been deleted on the host's side. Therefore, the agent only signals his owner that *he* shall become active.

Note that in the `last-reminder` case – as can be seen in fig. 4 – the message contains a text for non-CAP users *not* to respond by striking the reply key, but to contact the host in another way (see fig. 4).

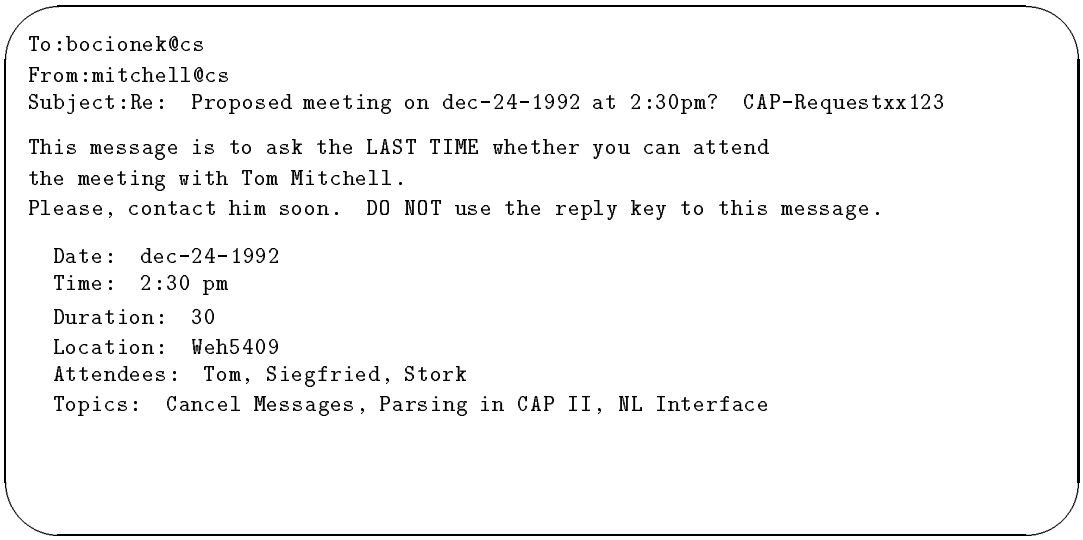


Figure 4: A last-reminder bid message as sent from the host to an attendee

2.3 The Automaton

In the following we describe the automaton, i.e. the operational concept of every CAP II agent. Two diagrams are given in figs. 5 and 6 that contain the states for all possible actions, set up of new meetings, moving of meetings, cancellation and exchange of information. The first picture characterizes the automaton on a host site, the second on an attendee site. This separartion is made for the sake of clarity only. In the implementation, there is only one automaton per agent that copes with all kinds of messages.

Note that only the *bold ovals* are the states that are visible to outside and on which new email messages from outside have any effect. All other states are intermediate and shown in the diagrams only in order to make the decisions of the agent more "visible". The two boxes, labeled `host-start` (or `att-start`) and `cancelled` are *explicit* final states of one negotiation. There are *implicit* finite states as well: `entered` on the host's site and `validated` on an attendee's site. Each event that is not cancelled eventually will reach that state and stay there (if no move or cancel occurs) until the event has passed and the time-out handler deletes it while regularily filtering old events out of the bookkeeping lists (see also section 2.2.5).

Fig. 5 shows that a host agent needs only three states. After a new initiative to set up a meeting (message type *usr-proposal*) the proposal is sent to all attendees and the state **proposed** is entered. The host agent stays in this state until enough *yes* bids have been received. Then it sends the confirmation and enters state **confirmed**. If eventually all validations of the attendees have come in the host agent finalizes the event, displays it in the calendar and goes to state **entered**.

All the intermediate states show the decisions the host agent has to make when move and

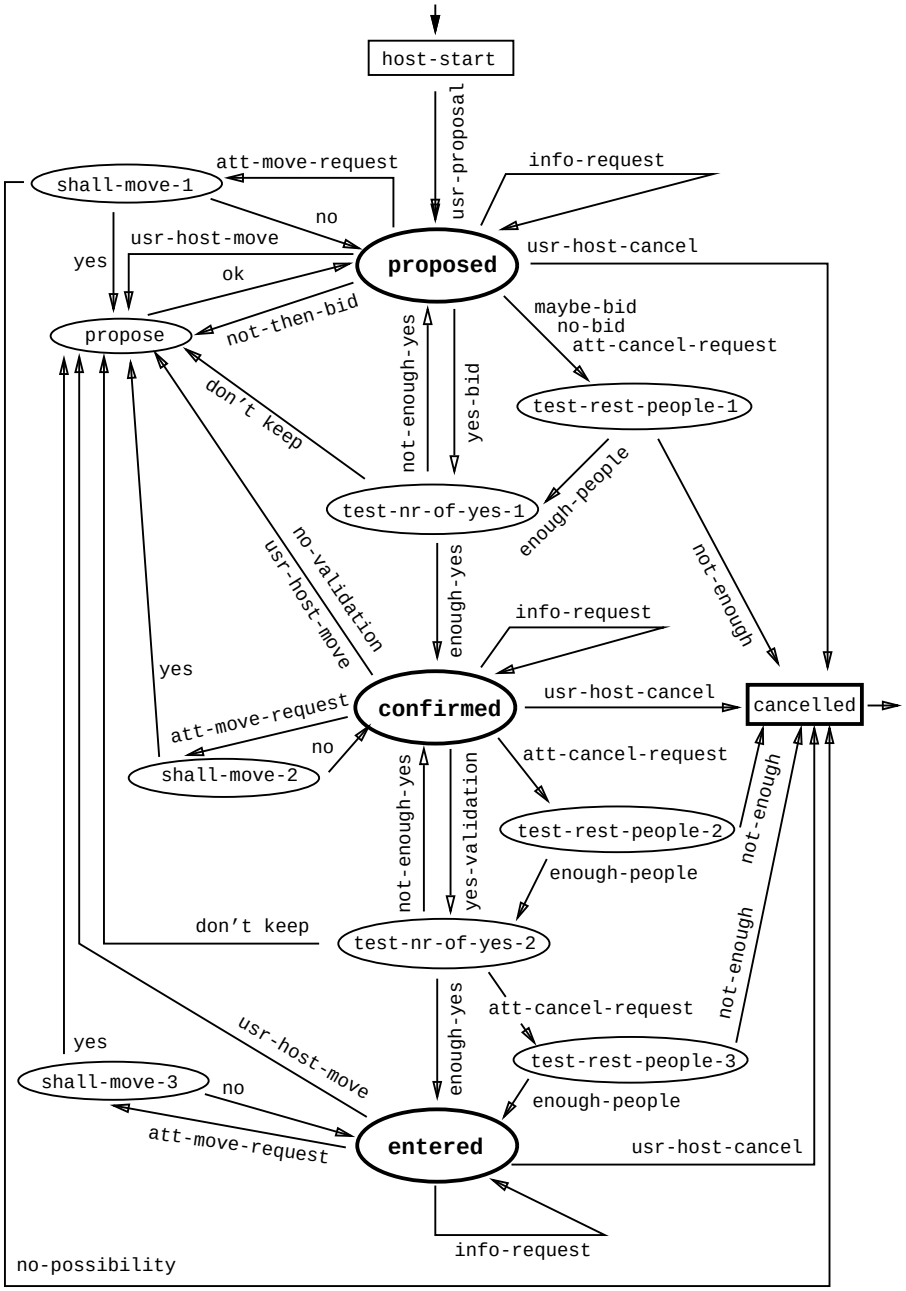


Figure 5: Transitions during an event negotiation on a host's site

cancel requests have to be processed, either from its own user or from an attendee. *Information-requests* can be received in any of the states but do not cause a transition.

As can be seen in fig. 6, the automaton on an attendee's site contains eight states. This fact is due to the "master-slave" relationship between host and attendees. Hosts can decide and

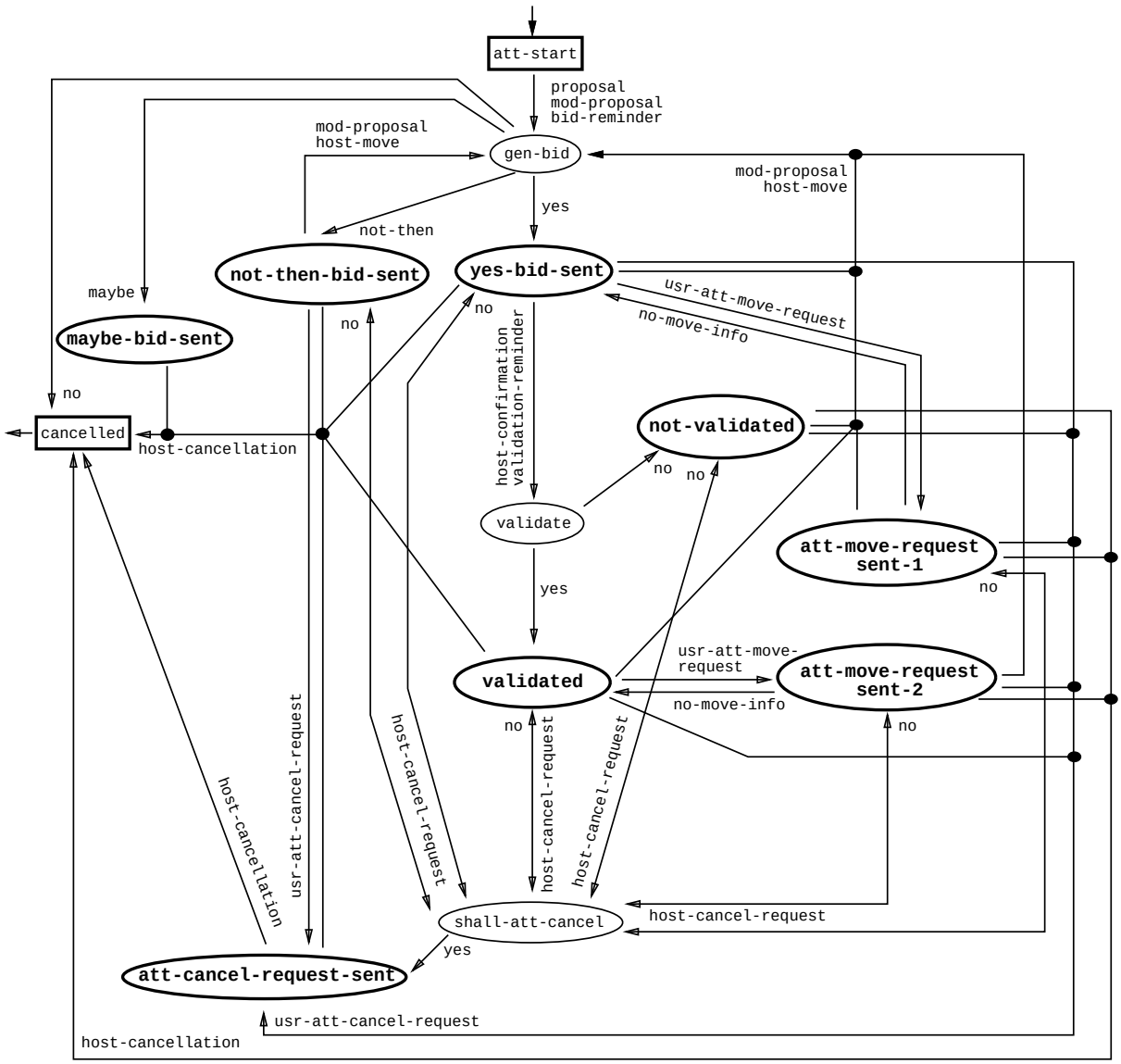


Figure 6: Transitions during an event negotiation on an attendee's site

initiate cancellations and movements of events "without discussion". An attendee has always to ask and wait (in an extra state) whether the host accepts the request or not. Additional (wait) states are also necessary for the different types of answers an attendee can send back as a bid or a validation.

If the whole negotiation works out straight forward, only the states **yes-bid-sent** and **validated** are used. In all states except **maybe-bid-sent** and **att-cancel-request-sent**, the attendee's agent can receive a *modified-proposal* or *host-move* that starts a new negotiation cycle by generating a new bid (in intermediate state **gen-bid**).

If the agent is in the **maybe-bid-sent** state, it plays the role of an informee. I.e., it might receive messages from the host but is not *asked* for anything.[15]

Note that, for sake of clarity, the messages types *information-request* and *host-information* are not shown in fig. 6. Since both do not cause a state transition they are possible in every state.

2.4 Discussion

As shown in the previous chapter, our CAP II design fulfills the requirements on the negotiation stated in section 2.1. The agents are able to negotiate meetings via email *autonomously* and disturbing the user as little as possible (requirement 4). CAP II is modeled according to people's behavior when doing that job.[16] It supports three sorts of parties, the host, attendees and informees. These three are the common actors when human beings cooperate by means of meetings. The possible – and in our opinion complete set of – actions are the setting up, moving and cancelling of events (see requirement 2 in section 2.1). Move and cancel services are also a comfortable means for the user to reset any decision or commitment of the agent (requirment 5 in section 2.1). In addition to move and cancel, requests can be sent to ask for information about a meeting, and information can be sent back.

The CAP II protocol goes even somewhat farther than human beeings operate when negotiating meetings. It provides a *two-way handshake* mechanism - i.e. the exchange of confirmation and validation messages - after an agreement was found upon a certain date and time. By this mechanism the whole range of commitment strategies can be realized (as enumerated in [18]). In particular – our requirement 3 in section 2.1 – weak commitments are possible that allow the agents to move events during a negotiation if more important requests are received that collide with the currently processed one.

The price is a quite complicated time-out handling (requirement 6 in section 2.1) because a host's agent has not only to *wait* for bids during the "pure" negotiation, but also for the validations at the end. However, the gain to allow the agents to deal with *tentative* times, is - in our opinion - worth the additional effort. The agents have now the ability to move events until they are finally confirmed, i.e to realize a *wait with blocking until confirmation* strategy as discussed in [18].

We believe that the finite automaton model as the basic processing concept of CAP II agents is *adequate* (see also requirement 1 in section 2.1) to describe the way human people act when setting up meetings. Every negotiation is always in a certain state, waiting for the next message that will determine the next action and the transition to the following state. The model can be extended quite easily if novel "behavior" has to be included. For example, an announcement of an interesting talk by a famous person via email could be read by the agent and lead to a move request for another meeting at the same time.[17] The request can be a *question* to the owner as long as the agent does not "know" for sure what its master would prefer. But if it once found

[15] In the current implementation there is no possibility to set the status of an informee back to that of an attendee.

[16] Note, that counter proposals are not fully implemented yet. They are necessary to support the full spectrum of possible actions performed by people.

[17] Features like those are *not* yet included in CAP II.

out - and the learning mechanism in CAP II supports such improvements - that, for example, Herb Simon's talks are considered to be a *must*, it can start negotiations to move "colliding" events with other agents even autonomously.

Compared to the literature we found, CAP II goes much farther in terms of autonomy and human-derived behavior than all other systems. This includes the concept as well as what has been implemented so far. CAP II also shows that the basic elements of every day's office work, as far as it concerns synchronisation activity via email, are only few. Moreover, they can be covered by operational concepts that are well-knowm in computer science (finite state machines in CAP II). We believe, therefore, that this approach can be used for additional tasks too, like room reservation or setting up an agenda for a visitor (a quite bothersome task as everybody knows who had done it once). The latter, indeed, is just a combination of meeting negotiation and room reservation, but with many people involved who all have their own calendars and schedules.

More powerful concepts, e.g. *Petri nets*, may be required for tasks like *stock administration* where customer orders, supplier messages and planning policies of the management have to be handled in parallel. Agent support could facilitate that job in many ways. The basic elements of such agents (like the user interface, action selection after email analysis or time-out handling) will be, in our opinion, quite similar to those of CAP II for meeting negotiation. First steps in that direction, i.e. using Petri nets as a concept for automated purchases handling, are reported in [11].

3 Including Humans Influences the Message Format

The second major objective of the CAP II project was the inclusion of both users with and without CAP II agent (see 1.1). This means the email messages have to be formatted in such a way that the agents are able to extract the necessary information, but also so that humans can comfortably and effectively participate in the negotiation as needed. The first subsection of this chapter will discuss the requirements on such an interface, as well as the "semi-structured" solution taken for CAP II.

After that we will describe the various messages that are to be exchanged by CAP II negotiations and their mapping to LISP data structures as necessary for the agents' work. The last subsection outlines some technical details of how CAP II handles the "parsing" of email messages. The major problem is that the messages are not constructed according to a formal, perhaps context-free, grammar which could be parsed by well-known and efficient algorithms. Fortunately, email messages for calendar negotiation are quite small, so the latter does not have to be a real handicap.

3.1 Semi-Structured Messages in CAP II

As mentioned briefly in chapter 1.1, several possibilities exist for organizing the message exchange in CAP II, from character-oriented formats to a pure natural language interface. Each of these has to be analyzed with our second major objective in mind, which specifies that the negotiation of meetings should include both CAP users and non-CAP users. In the next section we will enumerate a couple of finer-grained requirements - derived from the objective - and discuss our solution according to them.

3.1.1 Requirements and Solutions

The following requirements outline more precisely what it means to incorporate human participants and computer agents in negotiation processes based on the exchange of messages via email.

1. **Usual email:** The email text exchanged should be "as similar as possible" to those that human subjects "normally" write when setting up a meeting by those means.

2. **Simple filter mechanism:** Email concerning a meeting negotiation should be able to be filtered out from the stream of incoming email without time-consuming analysis of each message.

3. **Simple event recognition:** It should also be possible to determine which meeting is the subject of the message without complicated analysis of the whole text.

4. **Easy parsing:** Although the messages should be very "natural", it must be possible for every CAP agent to find the necessary information quickly and without complicated semantic analysis.

5. **Support for non-CAP users:** Special mechanisms should be provided to help non-CAP users write computer-understandable email messages correctly.

6. **Automatic completion:** Missing data in non-CAP user messages should be added by the receiving agents in cases where they are able to do so.

A pure natural language interface fulfills the first requirement perfectly, but violates the next three. Formatted records, on the other hand, are a contradiction to the first requirement that we want to provide "natural" email text. Therefore, we have chosen somewhat the middle of the two extremes: *semi-structured email*. The main portion of the messages are sentences as everybody would use for meeting negotiation. In addition, some keyword-value pairs provide the necessary information in a clearly arranged style. Fig. 7 shows an example of a meeting proposal generated by a CAP II agent. In the following, we will discuss our approach, using that example, and how far it meets the requirements stated above.

```
To:bocionek@cs
From:mitchell@cs
Subject:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123

This message is to suggest a meeting with Tom Mitchell
as follows.  Please let us know if such a meeting is
acceptable to you.

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface

You can make your reply computer-understandable by
beginning your message with one of the following words:
Yes, Not-then, Maybe, No
```

Figure 7: A meeting proposal generated by a CAP II agent

Requirement 1: Usual email
The subject line in fig. 7 contains a sentence that (or a slight variation of it) nearly everyone might use. The only artificial items are the word `CAP-Request` and the identifier `xx123` at its end. Both are used for filter and recognition purposes (see below).

The text "`This message is to suggest ...`" is also "as natural as possible." The word `suggest` carries the key information about the purpose of the message:[18] It indicates that the message must be a proposal or a modified proposal. In the case of a confirmation, the word `confirm` would appear at the same position in the first sentence (but without the answer suggestion at the end; see chapter 3.2 for a complete list of possible messages and their semantics).

Whether the keyword-value pairs, like `Date: dec-24-1992`, are necessary or whether they should be included in the text as in the subject line is, in our opinion, a question of taste. We have seen many people who write their meeting email in such a manner in order to clearly arrange the key information, and others who do not. But we feel that this style is not "unnatural" and hardly would bother people receiving it. Moreover, it facilitates the parse step to some extent and so we decided to choose it without further discussion (see the discussion of requirement 4 below, how date and time in the subject line are related to date and time in the text).

At the end of the message, a help text "`You can make your reply computer-understandable by ...`" is added to inform possible non-CAP users how to respond correctly (see discussion of requirement 5).

Altogether we believe that our way of generating meeting email in the style presented in fig. 7 will be accepted by people because it is quite short, it looks like what they are used to receiving and the semantics are clear. For CAP II agents, as we will show next, this semi-structured solution is also no problem.

Requirement 2: Simple filter mechanism

By a *filter mechanism*, we mean that some program must *separate* the email concerning negotiated meetings from all others coming in, and transmit them to the CAP II agent. *Simple* filtering, according to our deliberations, is a mechanism that is based on available operating system support and, in particular, does not need to analyze the complete email (which may be rather long in the case of messages not concerning meetings, say a large postscript file, for example).

We use the mechanism of a `maildelivery` file as provided by the `MMDF` package for email processing in the CMU Mach/Unix environment. We added a line

```
Subject     CAP-Request     |     A     cat >> newmail
```

to the file `.maildelivery` that recognizes the word `CAP-Request` in the subject line and appends the email to a file, say `newmail` (i.e., it "pipes" it – as indicated by the | character – to the UNIX `cat` command followed by the >> redirection sign). The CAP II agent's email subprocess polls this file and, if new email has arrived, reads and translates it into a LISP record as needed by the agent. Note, that the character `A` in the filter line says that a message shall be considered *delivered* once it has been appended to `newmail`. It will *not appear* in the user's regular mailbox (if desired, that could be provided by just using the character `R` instead).

In fact, the word `CAP-Request` is the only presupposition needed so that people with no CAP agent can participate in automated negotiations. If a CAP user receives an email *without* that keyword in his regular mailbox, and the message itself has some suitable form (i.e. words

[18] The position of the word `suggest` within the line does not matter.

like suggest or confirm appear), he can add `CAP-Request` easily to the subject line and forward it to himself. He can also add some necessary keyword-value pairs if this information cannot be found in the subject line. Even if the email has no acceptable form for the CAP agent, the user can forward it to himself (after adding `CAP-Request`). His agent will nevertheless handle the email properly (see below the discussion of requirement 5).

Requirement 3: Simple event recognition
Recognition of an event means that the CAP II agent must be able to find out to which meeting under negotiation a particular message is related. The basic means for that job is a so-called *event-id*, an identifier that is unique for every meeting under negotiation. It must be provided by the initiator of a meeting (when using CAP II, the agent does that job) and is appended at the end of `CAP-Request`, like `xx123` in fig. 7. CAP II agents use a combination of the host's name and a date/time stamp. However, this is not a must, all kinds of identifiers can be chosen (but they should, of course, ensure the necessary "uniqueness").

We feel that our mechanism is sufficient, even where non-CAP users are concerned, for two reasons:

- Generally, people construct a reply to a message by just hitting some `R` key or clicking at a response button. After that, the subject line of the email is *copied* into the reply, usually preceeded by `Re:`. Hence, if the event-id appeares in the received message, it will also be part of the response. We will outline in the discussion of requirement 4 how *inconsistencies* between subject line and the text of the message are treated.

- If a non-CAP user is the host and sends an initial proposal that contains `CAP-Request` but no event id, each CAP II agent receiving it constructs an identifier automatically and responds by using it. In the case of a two-party meeting no more problems occur, assuming the host uses the reply of the mail tool as described above.

  In the case of more than two parties we reason as follows: Since the host has no agent, he has to handle each message personally, i.e. *he* constructs the replies. He can do that *without taking care of event-ids* by means of the mail tool's reply mechanism and all will go right perfectly, even though each partner has probably generated a different id. What we want to avoid is the situation in which the non-CAP user, becoming aware of the "mess" with the ids, attempts to fix the problem by changing id numbers (see below the discussion of requirements 5 and 6).

  Note that, nevertheless, this problem (as many more concerning non-CAP users) is not fully solved. CAP II agents will work correctly "most" of the time if the so-called "good-will assumption" holds (see section 3.1.2).

Requirement 4: Easy parsing
Parsing, here, means to find out the key information about a meeting by analyzing the email message. *Easy parsing*, according to our deliberations, means that we do not want to use formal

algorithms, like LR parsing, or heuristic search techniques together with rules, lots of domain knowledge, context analysis etc., as would be necessary for a natural language approach.

Our semi-structured message format meets this goal completely: Searching for keywords like `"suggest"` (as in fig. 7) or `"confirm"` is sufficient. The definition of one different word for each type of message ensures a quick recognition of what the sender wanted to say. Matching of keyword-value pairs for date, time etc. yields the necessary data for a meeting. All recognition can be done by going through the text one time only (chapter 3.2 gives examples for all possible message types; chapter 3.3 discusses the parsing algorithm).

There may be *inconsistencies* between data in the subject line and in the text, i.e. the date and time information may differ. This inconsistency will usually come from the mail tool's reply mechanism that just copies the subject line. In the text, alternative data can occur. For example, an attendee might send a counter proposal back to the host. Since a (non-CAP) attendee is not expected to adapt the subject line, or even better is expected *not* to do it, CAP II agents have to handle this case.

Our approach works as follows: First, the agent tries to extract date and time from the subject line. If the keywords `"Date"` and `"Time"` are used in the text, the succeeding (and correctly parsable) values *overwrite* any previous date and time information. In the context of a `"How-about"` bid or a modified proposal those data are then interpreted as the new suggestion. The old data can always be recalled from the agent's bookkeeping lists if necessary. In addition, `"New-Date"` and `"New-time"` are used by CAP II agents in messages sent after changes. They help to focus a human's attention immediately on the important parts of a message. There will never be a `"Date"` *and* a `"New-Date"` field in *generated* messages (as there won't be for time or other keyword-value pairs). If a non-CAP user puts both in a message, the values of the `"New-"`fields are, obviously, assumed to contain the alternatively proposed data.

We believe that our semi-structured approach with keywords and keyword-value pairs allows CAP agents to parse the messages *easily* (in the sense formulated above). Furthermore, it allows us to *incorporate new types* of messages or make changes quickly. This became necessary very often during the development of CAP II, and it seems to continue. The reasons are mainly

- attempts to include new facilities (like sending information requests),
- attempts to make the agents more clever (i.e. parse even more "natural" text (for example, `"How-about tomorrow"` instead of giving an explicit date),
- attempts to extend the support for non-CAP users (see below) and
- different tastes of users that lead to suggestions whathow they like to receive meeting email.

It turned out that many new suggestions could be considered without problems once they were rated as worth being included. We expect that many more ideas will reach us as soon as CAP II finds wider use.

Requirement 5: Support for non-CAP users
The following functionality was included in our CAP II system to facilitate the participation of non-CAP users in meeting negotiations conducted by computer agents:

- One single keyword, namely `CAP-Request`, is sufficient to include email from non-CAP users in computer-processed negotiations. This word is quite easy to remember and, if forgotten or spelled differently, can be added by the receiving human as described above.
- Just *using the reply mechanism* of the underlying email tool is sufficient to fit into the filter and recognition processes of CAP II message exchange (see above).
- *Suggestions how to answer* at the end of a CAP generated email (see fig. 7) reduce the possibility that a non-CAP user will respond with a message that cannot be "understood" by an agent.
- *Help messages*, automatically sent back to a non-CAP user in the case of an unparsable response, improve the success of the agent's background work (see fig. 8).
- The *automated addition of missing information* is described in the discussion of requirement 6.
- Further support is given by some kind of *"tolerant" matching of keywords and values*. For example, a date value may be written as `"12/24/92"` but also as `"dec-24-1992"`, or the time as `"14:30"` or `"2:30 pm"`. Also the position of keywords in the text is not restricted (too much).

Our experience with CAP II so far has shown, that especially suggestions and samples were a key to the success of CAP II. Nobody has had to *learn* anything new. Samples exist for all message types used during a negotiation (proposals, bids, confirmations and validations). In addition, the mechanism to just hit a reply key in order to fulfill *all* necessities for the administration of events was rated as very convenient. We think, although further improvements are still possible, CAP II provides a useful interface that fulfills the requirement sufficiently.

Requirement 6: Automatic completion
The automatic completion of missing data in non-CAP user messages concerns two different issues:

- The completion of the messages itself in order to facilitate further transmission and
- the addition of event data that are missing (or cannot be parsed properly) in a message.

Only one mechanism is provided, so far, for the first case. Event-ids are generated by each CAP II agent that receives a proposal without one (see the discussion of requirement 3).

The second case is also easy to handle because the agent manages bookkeeping lists about all events. By means of the event-id, it immediately can retrieve any required data pertaining to the meeting in question. However, in the case of a modified proposal, it obviously *cannot* do that.

```
To:schwarzenegger@warner.com
From:mitchell@cs
Subject:ERROR IN CAP mail

=> THIS IS A COMPUTER GENERATED MESSAGE

An error occured in the bid that you sent.  A sample bid is
shown below.  Please, follow the format prescribed by it.

================================================================

                            SAMPLE BID
To:  mitchell@cs
From:  schwarzenegger@warner.com
Subject:  Re:Meeting on dec-24-1992 at 2:30 pm?  CAP-Requestsss23

Yes

================================================================

Note:  All parameters are optional.  The subject header MUST
CONTAIN the word CAP-Request together with the identifier at
the end.

Possible replies for a bid are:
‘Yes‘ - You accept the proposed meeting.
‘No‘ - You will not be able to keep the proposed meeting.
‘Not-then‘ - You wish to have a meeting, but at another date.
‘Maybe‘ - At the current time you cannot decide.
```

Figure 8: A help message as will be sent by CAP II when an unparsable email is received.

3.1.2 The "Good-will" Assumption

So far, we have presented and discussed the requirements of and our approach to a semi-structured message format for meeting negotiation that is suitable for both CAP II agents and unaided humans. However, it is necessary to mention that the success of our solution heavily depends on the *good will* of all participants. Since we do not provide a natural language interface - and deliberately don't want to (see above) - no semantic analysis of the email takes place, only "dumb" matching of keywords. Hence, it is easy to trick CAP II agents by, for example, including *nots* in the text like "`This message is not to confirm ...`" etc. In the case of agent-generated text, this cannot happen. In the case of text written by humans, one has to live with it.

We have significant experience in sending and receiving such messages using the current CAP system, and find that human recipients are willing and able to respond in a computer-readable fashion. In particular, while the current CAP system does not support automated negotiation,

it will (at the user's request) compose and send an invitation nearly identical to the example in fig. 7. After months of routinely sending such messages we find that the vast majority of recipients respond as requested. The others typically respond in free-text form, usually with a more complicated subject to convey than allowed by a one-word response. One of the open empirical questions to be answered by using CAP II is what proportion of user responses will fit into the message types provided by CAP II's model of negotiation.

Although some people will probably try out the game of confusing a CAP II agent *for fun*, we do not think it will be a problem in general. Our assumption is that people, usually, are *interested in a particular meeting taking place*. It makes no sense to "sabotage" the efforts of a colleague.

Moreover, we can imagine that the negotiation will proceed faster, with fewer failures and reminders, because answering is so easy and short. In fact, one only has to hit one key, add one word (in the case of a bid) or change one keyword (like suggest to confirm) most of the time. Therefore, one is less likely to postpone the response to "somewhat later," as everyone certainly has observed sometimes in her/his own behavior.

From the considerations in this section, we derive our confidence that the matching approach for the task of meeting negotiation via email is adequate. We also have showed that this approach is no obstacle to the provision of a "quasi-natural" language interface that is acceptable also for humans. In the next sections we will describe all message types used in CAP II, followed by a little description of how our email parser works in particular.

3.2 Messages and LISP Records in CAP II

In this section we describe the messages that are sent from and to CAP II agents in order to

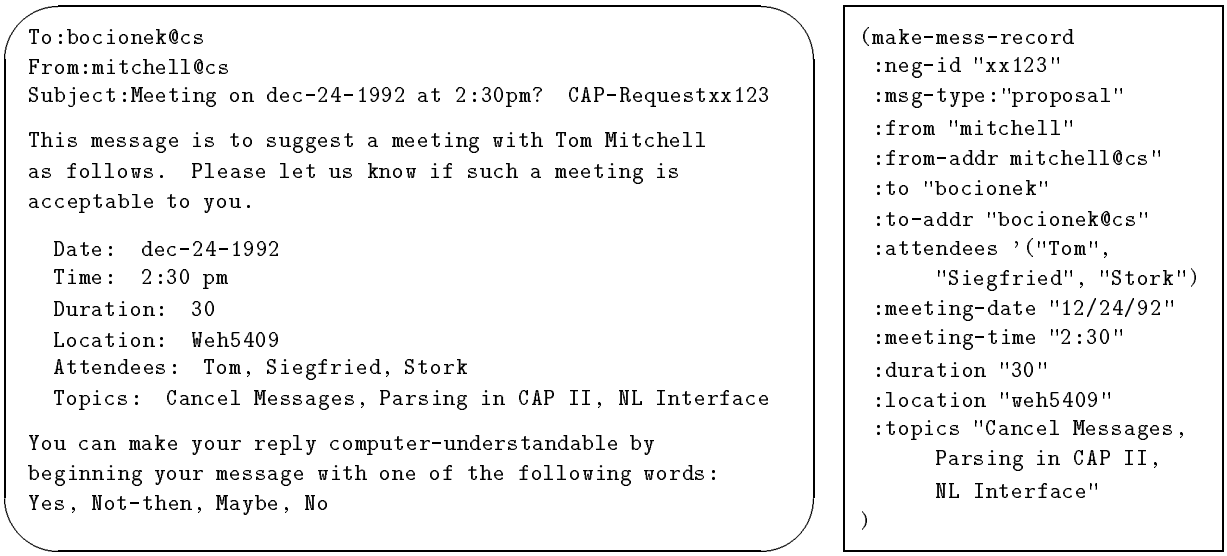
- setup new meetings,
- move meetings,
- cancel meetings and
- provide information and help.

For each message type we also give an example of the LISP element (an instance of structure `mess-record`) that is generated from it.

3.2.1 Setup New Meetings

To set up a new meeting the *host* sends a *proposal* to each of the intended *attendees* in which he includes the proposed date, time, duration, location, all attendees and the topics (see fig. 9). Note, that an initial proposal can be distinguished from every other CAP II email message because no reply sign (`"Re:"`) is at the beginning of the subject line. The second indicator for a proposal is the key word `"suggest"` as the fifth word in the message text.

A hint is given at the end of the text to show non-CAP users how to respond in order to allow CAP II agents to understand the message (see below for an example of a `Yes` bid).



```
To:bocionek@cs
From:mitchell@cs
Subject:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123

This message is to suggest a meeting with Tom Mitchell
as follows.  Please let us know if such a meeting is
acceptable to you.

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface

You can make your reply computer-understandable by
beginning your message with one of the following words:
Yes, Not-then, Maybe, No
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"proposal"
 :from "mitchell"
 :from-addr mitchell@cs"
 :to "bocionek"
 :to-addr "bocionek@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
)
```

Figure 9: Initial proposal sent from the host to each attendee

As an answer to a proposal, the attendees send a *bid* to the host. We have completely implemented an extended version of the *yes-no strategy* for bids as described in [18]. In this extension, we accept more than just `Yes` (meaning, "I can come at the proposed date") and `No` (meaning, "I cannot come at the proposed date; please, propose another one") as an answer. In CAP II, we allow:

- `Yes`: This bid means that the attendee (or his agent) has agreed to the proposed date and time and marked the event in his calendar as *tentative* (see below, when it is eventually fixed).
- `Not-then`: This is handled like the *no* bid in [18], i.e. the host has to propose a new date.
- `Maybe`: As mentioned before, the attendee indicates that he cannot decide yet whether the date is acceptable for him or not. Hence, the host puts this attendee on the list of "informees", so the attendee is notified about all ongoing changes, but no longer involved in the negotiation (see also 2.2.1).
- `No`: This bid means that the attendee will not come to the meeting for whatever reason. The host agent deletes him from the initial list of attendees. If he was the only attendee, the host agent deletes the complete event

Fig. 10 shows a CAP-request message with a `Yes` bid. Note that this `Yes` (like all other bids) must be placed as the first word of the message body as suggested in the proposal (see fig. 9 above). The word "`confirm`" in the text has no further semantics once the agent has recognized

the `Yes` bid. A `"confirm"` without a previous `Yes` bid, however, would indicate a *confirmation* (see below).

```
To:mitchell@cs
From:bocionek@cs
Subject:Re:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123

Yes,
This message is to confirm that I have the following
meeting with Tom Mitchell:

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"bid"
 :from "bocionek"
 :from-addr bocionek@cs"
 :to "mitchell"
 :to-addr "mitchell@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "yes" )
```

Figure 10: A Yes bid sent from an attendee to the host

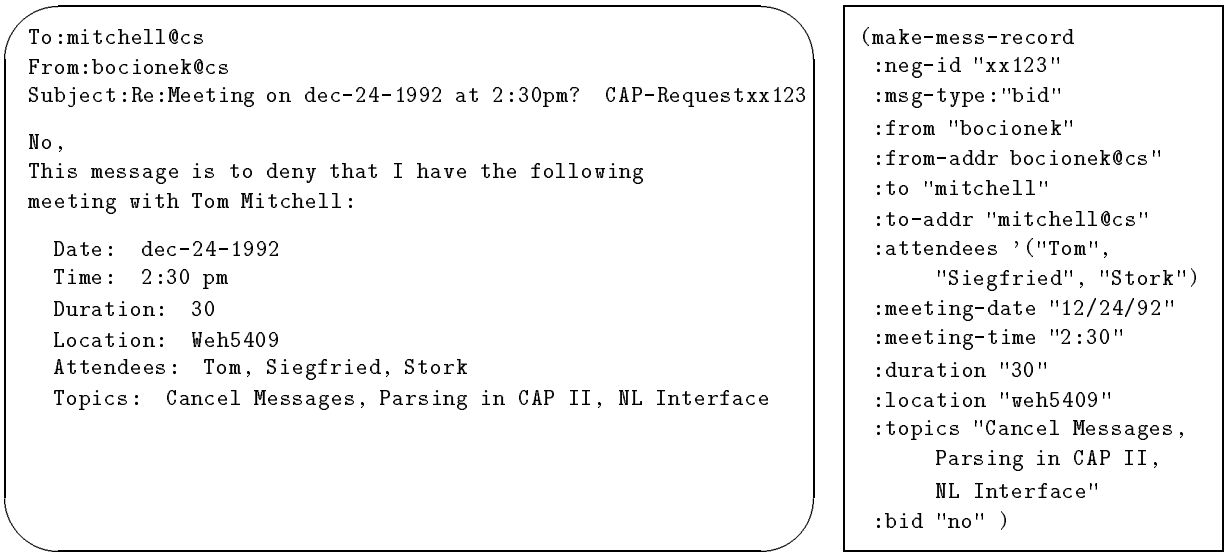
Fig. 11 shows a CAP-request message with a `No` bid. Note that `No` must be placed as the first word of the message body, analogous to 10. The meaning-carrying word `"deny"` in the text body also has no further semantics for the CAP agent.

Fig. 12 shows a CAP-request message with a `Not-then` bid. Note that `Not-then` must be placed as the first word of the message body, analogous to 10. The meaning-carrying words `"indicate"`, `"wish"` and `"another date"` in the text body are important to make the mesage understandable for the human, but not necessary to be interpreted by the CAP agent.

Fig. 13 shows a CAP-request message with a `Maybe` bid. Note that `Maybe` must be placed as the first word of the message body, analogous to 10. Again, the meaning-carrying words `"indicate"` and `"cannot-yet-decide"` in the text body are important to make the message understandable for the human, but not necessary to be interpreted by the CAP agent.

A second strategy, that allows CAP II agents to send so-called *counter proposals* as bids is currently implemented. Here, attendees have the possibility of actively proposing alternatives to a meeting date or time instead of only saying *Not-then*. The advantage of counter proposals is that the number of negotiation cycles may be reduced, because the host now gets knowledge about free time slots in the attendees' calendars. However, a scheduler is necessary for that task to solve the constraint problem given by the host's proposal and the set of counter proposals received as bids (see chapter 6 for a further discussion of this point).

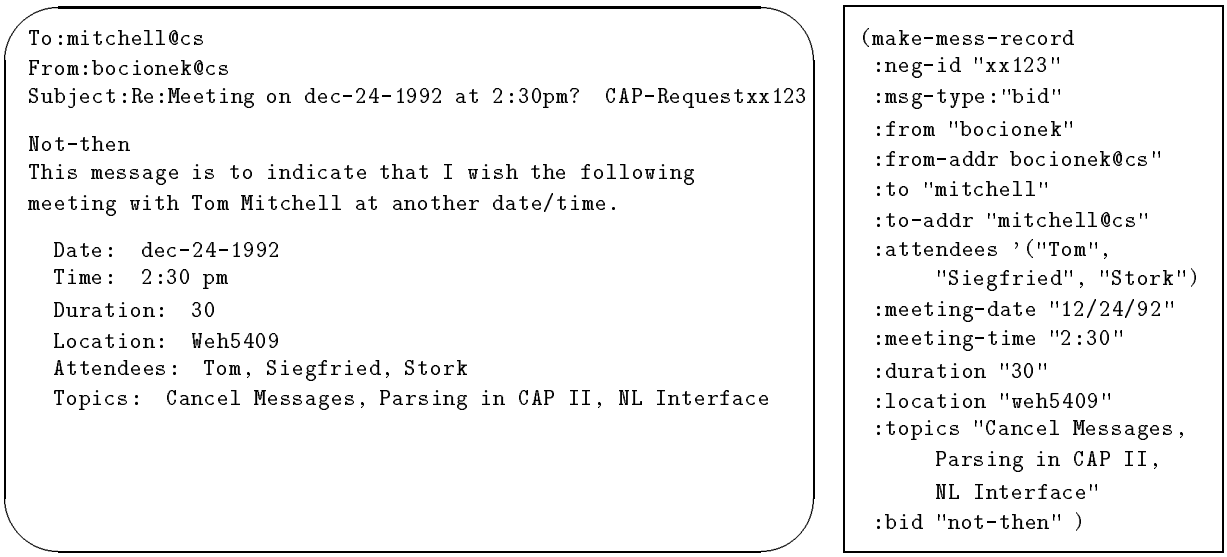
Fig. 14 shows a CAP-request with a *counter proposal* bid. Note that `How-about` in the first line of text refers to a counter-proposal bid. If the message was generated by an agent, it uses



```
To:mitchell@cs
From:bocionek@cs
Subject:Re:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123

No,
This message is to deny that I have the following
meeting with Tom Mitchell:

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"bid"
 :from "bocionek"
 :from-addr bocionek@cs"
 :to "mitchell"
 :to-addr "mitchell@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "no" )
```

Figure 11: A No bid sent from an attendee to the host

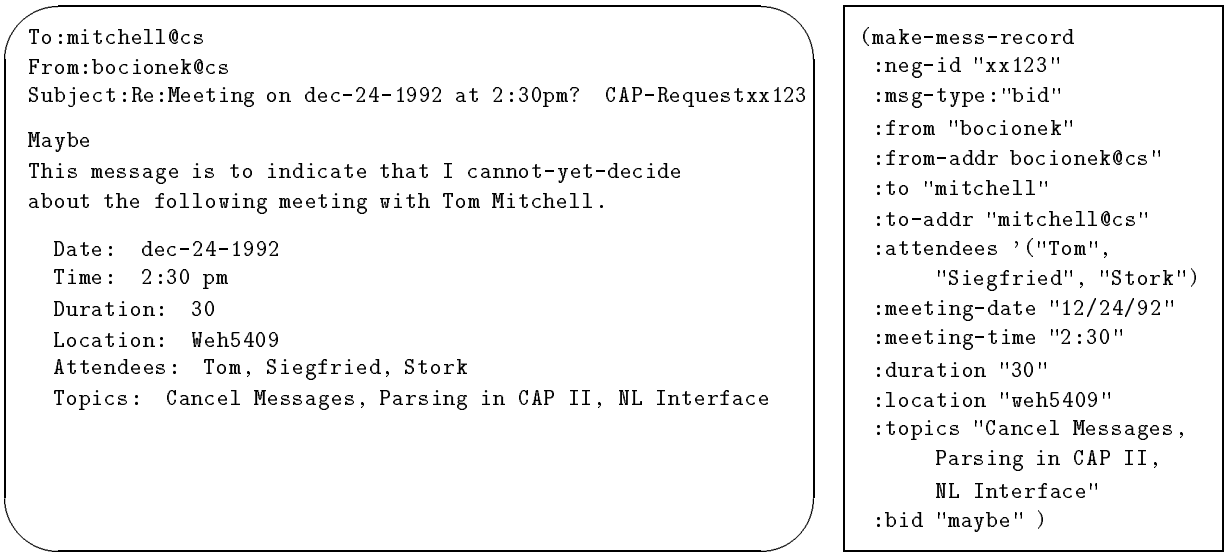


```
To:mitchell@cs
From:bocionek@cs
Subject:Re:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123

Not-then
This message is to indicate that I wish the following
meeting with Tom Mitchell at another date/time.

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"bid"
 :from "bocionek"
 :from-addr bocionek@cs"
 :to "mitchell"
 :to-addr "mitchell@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "not-then" )
```

Figure 12: A Not-then bid sent from an attendee to the host

`New-` keywords like `New-time` to focus the attention of the reader to the *new* data of the counter proposal. These `new-` keywords are not absolutely necessary. A CAP agent will interpret the message correctly even if a non-CAP sender used only the regular words `Date` and `Time` etc.

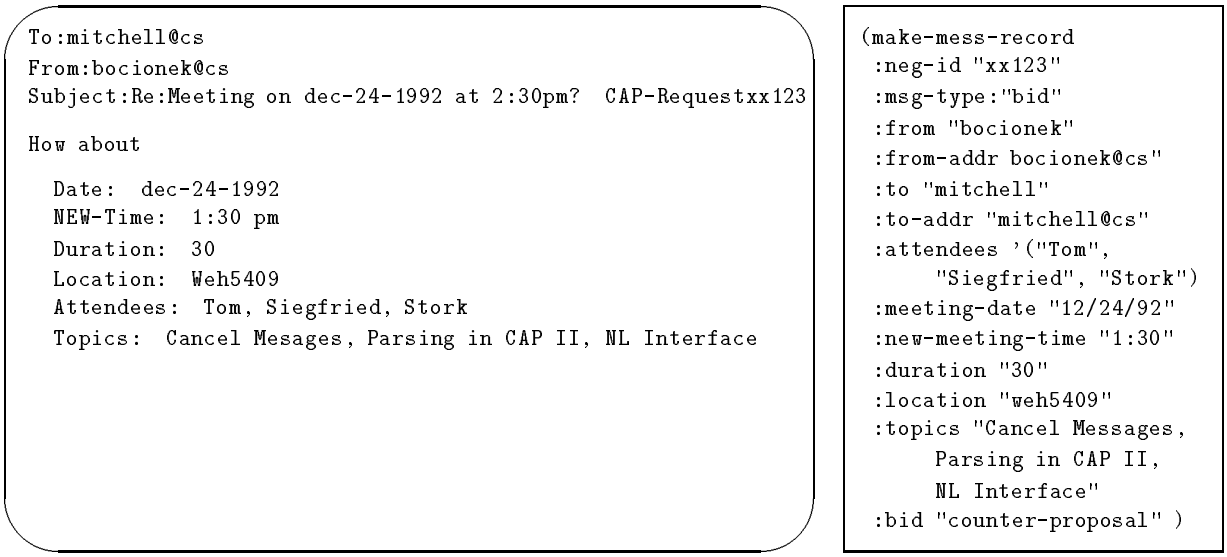


```
To:mitchell@cs
From:bocionek@cs
Subject:Re:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123

Maybe
This message is to indicate that I cannot-yet-decide
about the following meeting with Tom Mitchell.

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"bid"
 :from "bocionek"
 :from-addr bocionek@cs"
 :to "mitchell"
 :to-addr "mitchell@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "maybe" )
```

Figure 13: A Maybe bid sent from an attendee to the host

(see section 3.1.1, requirment 4).



```
To:mitchell@cs
From:bocionek@cs
Subject:Re:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123

How about

  Date:  dec-24-1992
  NEW-Time:  1:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Mesages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"bid"
 :from "bocionek"
 :from-addr bocionek@cs"
 :to "mitchell"
 :to-addr "mitchell@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :new-meeting-time "1:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "counter-proposal" )
```

Figure 14: A counter-proposal bid sent from an attendee to the host

Once the host has received positive bids (yes-bids) from all attendees, it sends a confirmation

message to each of them. Fig. 15 shows such a *confirmation*. Note that the keyword "confirm" must be present in the text of the message.

```
To:mitchell@cs
From:bocionek@cs
Subject:Re:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123


This message is to confirm that I have the
following meeting with Tom Mitchell.

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"host-confirmation"
 :from "mitchell"
 :from-addr "mitchell@cs"
 :to "bocionek"
 :to-addr "bocionek@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :confirmation "yes" )
```

Figure 15: A confirmation sent from an host to the attendee

After an attendee receives a confirmation of the host, its agent tests whether it can still keep the date and, if so, sends back a positive *validation* to the host. If not, the agent sends back a negative validation.

Fig. 16 shows a positive *validation*, indicated by the word "validate" in the text body. The validation is necessary to complete the two-way handshake in our protocol as described in section 2. Please note that one may type "un-validate" in order to indicate that one cannot validate the meeting.

3.2.2 Moving of Meetings

Once a meeting has settled, every party has the right to try to reschedule to a more convenient date. But the rights for host and attendees are different. If the host sends a *host-move-request*, the meeting will definitely be moved. However, the date and time will be negotiated as with all other proposals (for details see section 2.2.2).
Fig. 17 shows a CAP-request message of the type *host-move-request*. This is when a host moves an existing meeting to a new date, new time, etc. The meaning-carrying phrase in the text is "move the meeting".

The phrase indicates also that the host sent the message. If an attendee wants to *ask* for a movement, he has to use request a move instead (see below).

```
To:mitchell@cs
From:bocionek@cs
Subject:Re:Meeting on dec-24-1992 at 2:30pm?  CAP-Requestxx123


This message is to validate the
following meeting with Tom Mitchell.

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"att-validation"
 :from "bocionek"
 :from-addr bocionek@cs"
 :to "mitchell"
 :to-addr "mitchell@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :validation "yes" )
```

Figure 16: A validation sent from an attendee to the host

```
To:mitchell@cs
From:bocionek@cs
Subject:  Move Meeting on dec-24-1992 at 2:30pm?
CAP-Requestxx123

This message is to move the meeting with Tom Mitchell to:


  Date:  dec-24-1992
  NEW-Time:  1:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"host-move-request"
 :from "mitchell"
 :from-addr "mitchell@cs"
 :to "bocionek"
 :to-addr "bocionek@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :new-meeting-time "1:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "unknown" )
```

Figure 17: A move request sent from an host to the attendees

If an attendee sends an *attendee-move-request* to the host, the host decides whether to move or not.

Fig. 18 shows a CAP-request message of this type, i.e. an attendee requests the host to move

a meeting to another date, time, etc. This is indicated by the phrase "request a move" in the text body. We also accept the words "request" and "move" if they are found in one line.

```
To:mitchell@cs
From:bocionek@cs
Subject:  Move Meeting on dec-24-1992 at 2:30pm?
CAP-Requestxx123

This message is to request a move of the
meeting with Tom Mitchell to:.

  Date:  dec-24-1992
  NEW-Time:  1:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"att-move-request"
 :from "bocionek"
 :from-addr bocionek@cs"
 :to "mitchell"
 :to-addr "mitchell@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :new-meeting-time "1:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "unknown" )
```

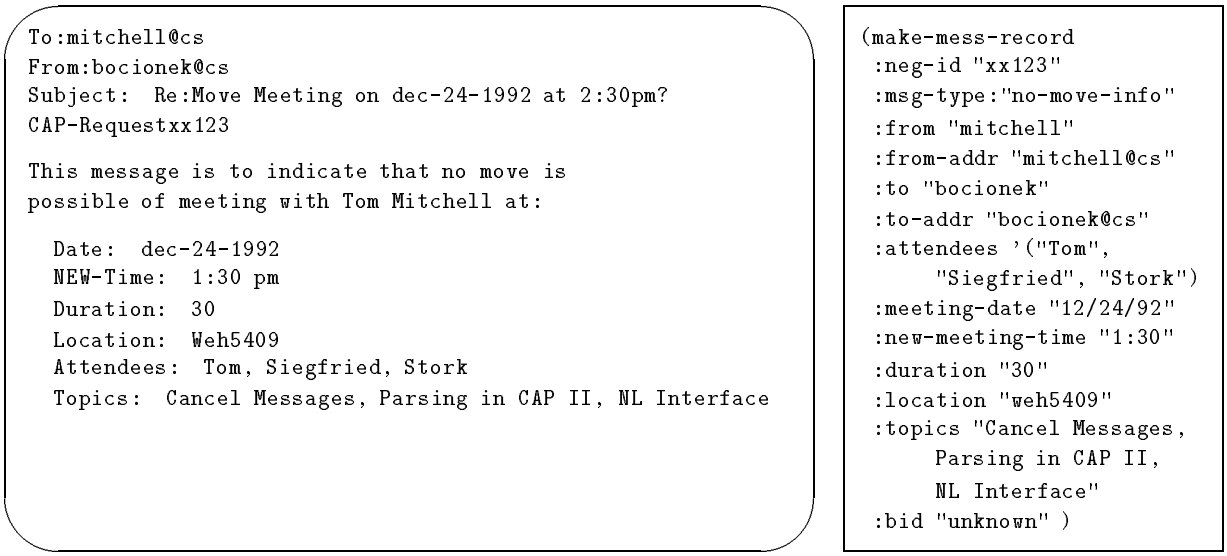
Figure 18: A move request sent from an attendee to the host

If the host decides *not* to move the meeting, its agent sends a *no-move-information* message to the requesting attendee. Fig. 19 shows a CAP-request message of that type. It is sent by the host to the attendee upon receipt of a move-request to indicate that no mov of the meetinge is possible. The meaning-carrying phrase is "indicate that no move is possible" in the text body. We also accept "no move" if those two words appear on one line.

3.2.3 Cancellation of Meetings

As described in section 2.2.3, there are means in CAP II to cancel meetings. The first possibility is a *host-cancellation*, sent from the host to all attendees. Here, there is no more negotiation, the event is cancelled.

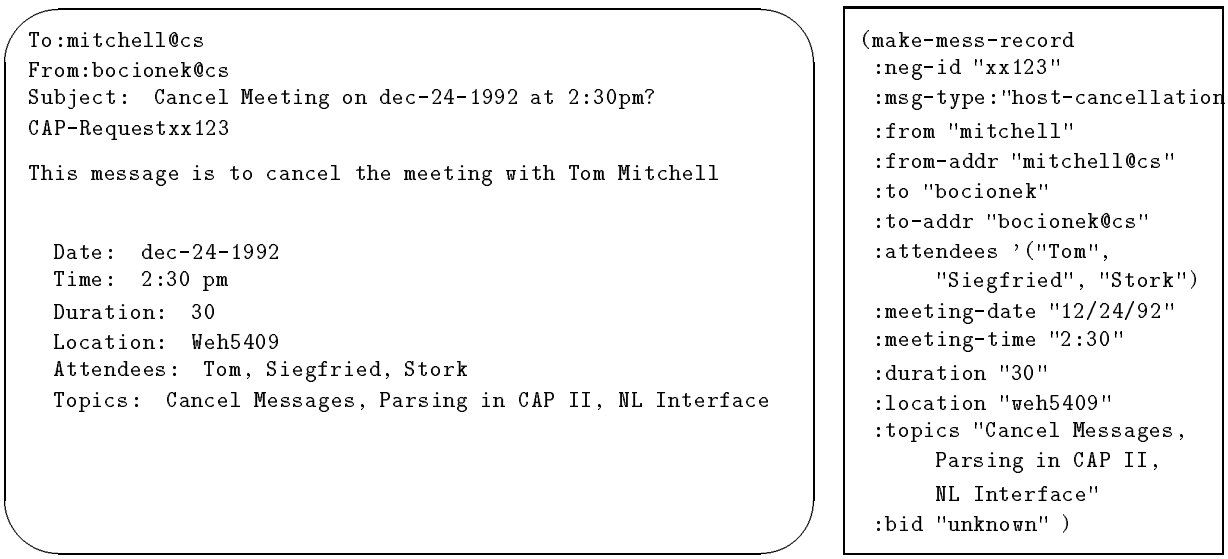
Fig. 20 shows a CAP-request message of type *host-cancellation*. This is sent from the host to the attendees indicating the cancellation of the meeting. The meaning-carrying word is "cancel", in the subject line as well as in the text body.
Both parties, attendees as well as host, can announce that they are not able to attend a meeting, but that the meeting shall be held as planned (for details see section 2.2.3). For that purpose, they have to send a *cancel-request*. Fig. 21 shows a CAP-request message of this type. The meaning-carrying phrase is "cannot attend" in both, subject line and text body. It is also acceptable, if both words are found on one line (so that a non-CAP user does not have to take care of the number of blanks, for example).



```
To:mitchell@cs
From:bocionek@cs
Subject:  Re:Move Meeting on dec-24-1992 at 2:30pm?
CAP-Requestxx123

This message is to indicate that no move is
possible of meeting with Tom Mitchell at:

  Date:  dec-24-1992
  NEW-Time:  1:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"no-move-info"
 :from "mitchell"
 :from-addr "mitchell@cs"
 :to "bocionek"
 :to-addr "bocionek@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :new-meeting-time "1:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "unknown" )
```

Figure 19: A no-move sent from an host to the attendee



```
To:mitchell@cs
From:bocionek@cs
Subject:  Cancel Meeting on dec-24-1992 at 2:30pm?
CAP-Requestxx123

This message is to cancel the meeting with Tom Mitchell


  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"host-cancellation"
 :from "mitchell"
 :from-addr "mitchell@cs"
 :to "bocionek"
 :to-addr "bocionek@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "unknown" )
```

Figure 20: An cancelation is sent from an host to the attendee

3.2.4 Exchange of Information about Meetings

After a meeting has settled, or in the case of changes - e.g., of the location - the host sends *information* messages to the *informees* (see also section2.2). The same messages are sent to

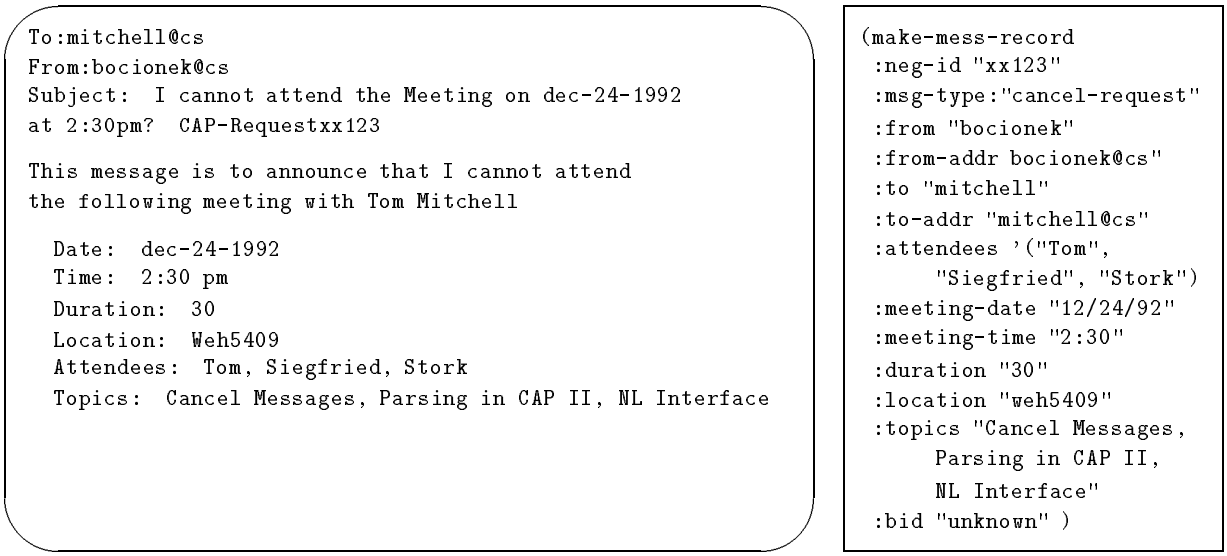


Figure 21: A cancel request as can be sent from attendees and the host

everyone who has requested information explicitly (see below).

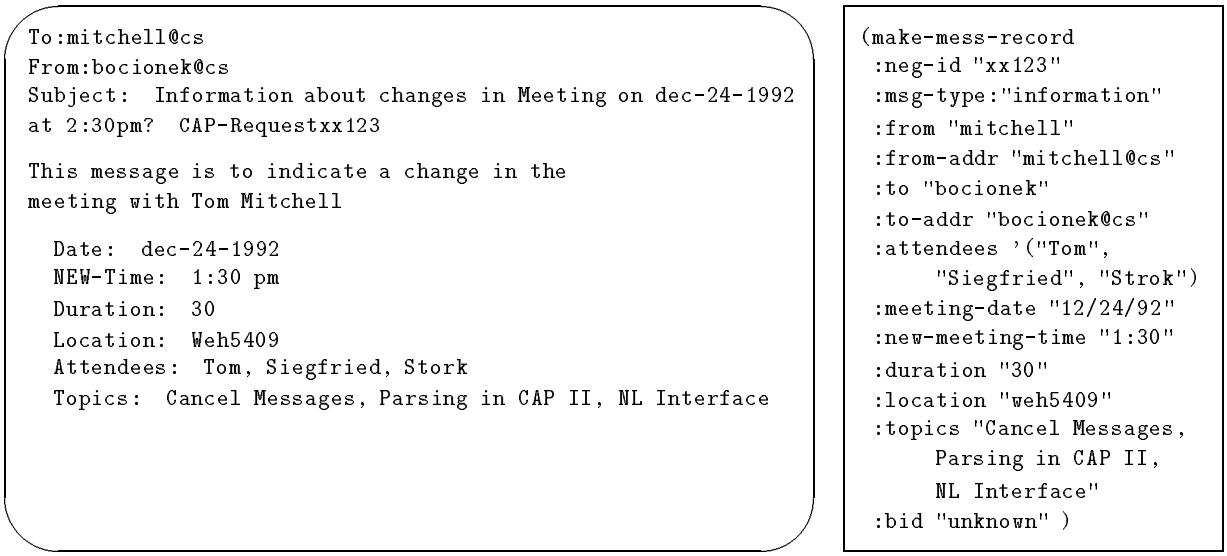
Fig. 22 shows a CAP-request message of type *information*. The meaning-carrying phrase is `"indicate a change"` in the text body.

The way how attendees or informees can request information about an event is shown in Fig. 23. This figure shows a CAP-request message of type *info-request*. The meaning-carrying phrase is `"ask for information about"` in the text body. Note that the data in the message is that which the attendee knows so far, but which does not need to be correct. The data fields could also be empty if the attendee has no complete knowledge about the meeting.

3.3 Parsing of Semi-Structured Email Messages in CAP II

The previous section has shown various messages that can be exchanged by CAP and non-CAP users. It is the job of the CAP II communication interface to map those messages to LISP records, so that the CAP II agents can perform the desired negotiation. We call these mapping procedures a "parser", although we only use searching and matching functions and no algorithms like those in computer language compilers.

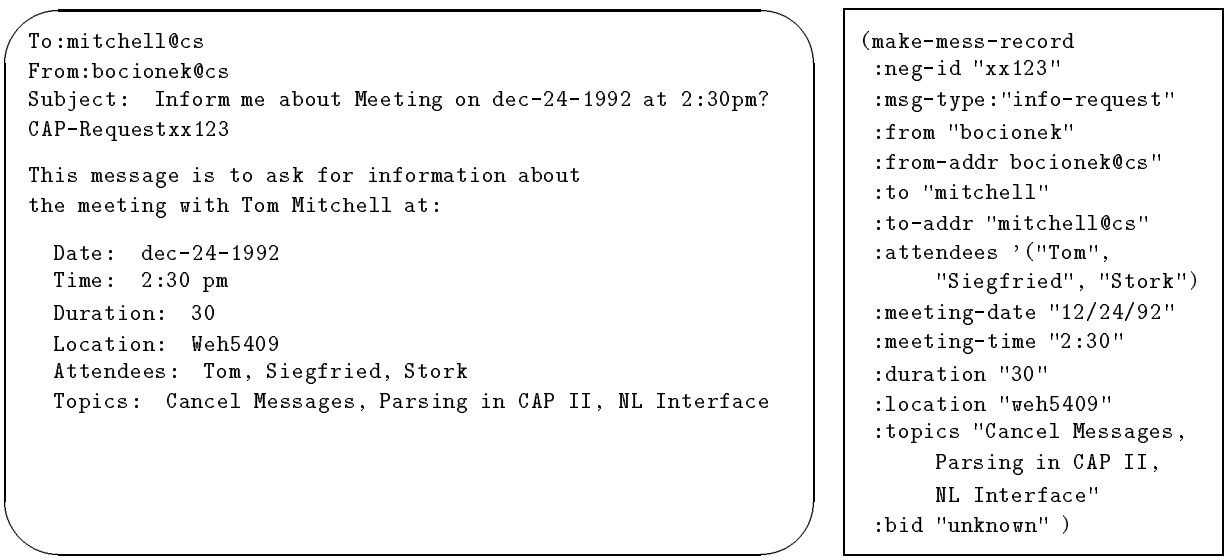
Algorithms of that kind - like LR - cannot be applied for the simple reason, that the email messages are not generated according to a regular or perhaps context-free grammar. Instead, we used some kind of "natural" language for the textual part (the motivation is discussed in detail in section 3.1), and keyword-value pairs for the determining data of a meeting (date, time, duration, location, attendees and topics). In addition, we introduced special keywords at special positions in the text to convey the information about agreement, confirmation, validation etc. (see section 3.2).



```
To:mitchell@cs
From:bocionek@cs
Subject:  Information about changes in Meeting on dec-24-1992
at 2:30pm?  CAP-Requestxx123

This message is to indicate a change in the
meeting with Tom Mitchell

  Date:  dec-24-1992
  NEW-Time:  1:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"information"
 :from "mitchell"
 :from-addr "mitchell@cs"
 :to "bocionek"
 :to-addr "bocionek@cs"
 :attendees '("Tom",
      "Siegfried", "Strok")
 :meeting-date "12/24/92"
 :new-meeting-time "1:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "unknown" )
```

Figure 22: An information indication is sent from the host to attendees or informees



```
To:mitchell@cs
From:bocionek@cs
Subject:  Inform me about Meeting on dec-24-1992 at 2:30pm?
CAP-Requestxx123

This message is to ask for information about
the meeting with Tom Mitchell at:

  Date:  dec-24-1992
  Time:  2:30 pm
  Duration:  30
  Location:  Weh5409
  Attendees:  Tom, Siegfried, Stork
  Topics:  Cancel Messages, Parsing in CAP II, NL Interface
```

```
(make-mess-record
 :neg-id "xx123"
 :msg-type:"info-request"
 :from "bocionek"
 :from-addr bocionek@cs"
 :to "mitchell"
 :to-addr "mitchell@cs"
 :attendees '("Tom",
      "Siegfried", "Stork")
 :meeting-date "12/24/92"
 :meeting-time "2:30"
 :duration "30"
 :location "weh5409"
 :topics "Cancel Messages,
      Parsing in CAP II,
      NL Interface"
 :bid "unknown" )
```

Figure 23: A information request sent from an attendee to the host

Because of the shortness of the email messages, the way they are parsed may not be so impor-

tant for efficiency reasons. Searching through the text several times would not cost that much.[19] However, as it turned out during the implementation, the message formats were always changing - or - subject to constant change. This included especially extensions and specific cases that had to be integrated step by step. Furthermore, it appears that this process will continue in order to make CAP II agents more capable and provide more "naturalness" for the different messages. In addition, the implementation of further agents for other tasks in the office automation domain, (see 6), which include email processing, is planned. For those new agents the already implemented email parser should be applicable as much as possible, or at least easily adaptable.

The considerations above lead to the following structure of the email parser in CAP II. The parser first reads in the whole message as a list of strings, and then approaches the problem of distinguishing the valid data from the invalid. In order to do this the parser breaks the message into two parts - the header and the free-text part. The first blank line after the header in the message file indicates the commencement of the free-text, and the parser essentially looks for this in order to separate out the two lists of strings.

1. **Parse Headers:** This part of the parser looks for keywords such as `To`, `From` and `Subject`. It then reads in the data present in these fields. One of the key fields is the `Subject` field. It should contain the name of the person sending the email message, the date and time of the meeting, and the event-id. The parser can handle dates like Jan-16-92, 1-16-92, 12/3/1992, and times like 12:18, 12.18, 23.36 (In case the time appears in 24 Hr clock form, the parser converts it to 12 Hr clock.) For the `To` and `From` fields it picks up the text after the colon and inserts it into the data structure. It is important to note that, in order for the calendar to receive these messages as CAP messages, the keyword `CAP-request` has to be in the subject line so that `maildelivery` can appropriately channel these mail files.

2. **Parse Text:** The text parser parses the email message after the headers. It looks for keywords in the text which indicate the type of message it is, and then scans backward and forward, in order to confirm this. Although the parser currently does not handle past and future references like "Can I meet you TOMORROW," we will try to make it more versatile in the future. Today it looks for the type of bid on the first line of text, which can be one of the following: `Yes, No, Not-then, Maybe, How-about`. In addition, it looks for keywords to distinguish the type of message involved. For example, `"confirm"` in the text would mean a *confirmation*; `"validate"` or `"un-validate"` would mean a positive or negative *validation*; and `"I cannot attend"` would indicate an *attendee-cancel-request*. In addition to parsing the type of message, we parse out keywords that have a colon to indicate the data, and are on a line by themselves. These include location, date, time, duration, topics, attendees, and new-location, new-date.

The messages can contain any other text that the parser will ignore. It only searches for a list of keywords, and upon finding the right set of them, decides on the type of message. If the sender of the email made a mistake, the parser tries to figure out what type of message it was,

[19] In fact, this was exactly the approach in the first, "hacked" version of the communication interface.

i.e. proposal, bid, confirmation, validation, etc, and then sends a sample message of that type back to him. Since the message formats are slightly complicated, it is possible to obtain help by sending a non-CAP user mail with `"Help on CAP-Request"` in the subject line.

3.4 Discussion

The second major objective stated in chapter 1.1 was to let people be involved in meeting negotiation even if they do not possess a CAP II agent. This objective was refined into a couple of more specific requiremants. As the examples of the last section showed, the requirement to provide a "usual" textual email interface was - in our opinion - met. This requirement concerned mainly the passive, perceptional part of user interaction, e.g. reading email.

When writing the response, non-CAP users have to perform only little work. Mostly, hitting the reply key and typing in one word is sufficient. In addition, non-CAP users are supported by comfortable mechanisms like a *suggestion line* in agent-generated messages how to respond correctly, or automatically sent-back sample messages if a non-CAP user message could not be parsed by the receiving agent. Furthermore, mechanisms for filtering and recognition of messages are provided in CAP II that require a minimum of non-CAP user interaction. Hitting the return key ensures both the presence of the necessary keyword `CAP-Request` as well as the return of the event identifier. Participants without CAP II agent do not need to care about that at all.

With our CAP II implementation we were also able to show that the necessary email parsers can work successfully by applying matching algorithms *only*. No sophisticated semantic analysis is required as would be the case with a completely unrestricted natural language interface. By enhancing the CAP II agents with a little *dialog-based knowledge acquisition component* [2] one can make the email parsing even more tolerant so that it can accept nearly *all* kinds of messages related to a certain domain like meeting negotiation. In chapter 6 we will outline this idea in more detail.

We believe that the success of quite simple matching, in addition to interactive user inclusion, is a more general result than only the adequate solution for meeting negotiation tasks. It should be possible to apply the method to *all kinds of office work* that is based on the exchange of email messages. For users without agents the message text would appear as they are used to receive them. Responding correctly should not require any additional training or other effort.

What we still need, nevertheless, are field tests in order to evaluate the acceptance of CAP II. Do people really trust an agent that acts on their behalf? Do we have to provide some levels of autonomy for the agents that can be controlled by the user (realized by "do all automatically" or "ask for permission" command options, for example)? How good can non-CAP users understand the suggested answers to a message? Is it intuitively clear for them what a "Maybe" bid means? Can they ask for further explanation? All those questions cannot be answered without some big testing effort. The testbed for these experiments, CAP II, is ready to go.

4 The Implementation

In this section we want to describe some details of our CAP II implementation, i.e., the architecture, the interface between the original CAP and CAP II, and the extensions of the user interaction in order to benefit from the CAP II negotiation capabilities.

4.1 The Architecture of CAP II

CAP II is implemented in Common LISP on a Mach/Unix OS platform. It provides - as the original CAP did [9] - a calendar interface as an EMACS process. The EMACS interface process has access to the CAP II agent by a remote procedure call (RPC) like mechanism with blocking send. On the other hand, a CAP II agent that asynchronously received an email message must be able to modify the CAP data and the calendar display, too. This can be done by a special communication interface (described in 4.2.1).

The CAP II agent consists of four sub-processes. As shown in fig. 24, `apprentice` is the interface to the Theo database with the learned rule set and to the calendar data. This sub-process works as in the first CAP version.

At the end of each interface command (like `add-event`) we put a procedure call that converts the command parameters to the required CAP II record (of type `usr-...`) and enqueues that record into the shared global message queue, i.e. at the *front*[20] (see section 2.2.4 for a discussion of that point). From there, the sub-process `automaton` can dequeue the message and processes it as described in chapters 2 and 3. If the processing of a message requires to send email, `automaton` just calls a LISP function `send-email` for that job, but not another sub-process. Note that putting the message into the queue can be considered as a *non-blocking* call to `automaton` by `apprentice`.

A second way to provide `automaton` with messages is the email interface sub-process called `email-watcher`. It polls the user's `newmail` file every 30 seconds (see requirement 2 in section 3.1.1). If new mail has arrived, it is parsed, transformed to a LISP record, and enqueued at the *back* of the shared global message queue (see fig. 24). If more than one CAP-Request message was found in the file `newmail`, the sub-process `email-watcher` copies all but the first one into a file `workmail` in order to buffer it there. After that this file is always processed first until it is emptied, after which messages are again taken from the `newmail` file. The implementation of the "CAP-Request filter" by means of a `maildelivery` file was already described during the discussion of requirement 2 in section 3.1.1.

The fourth sub-process is the `time-out handler`. It shares some global knowledge with the `apprentice` and with the `automaton` processes. When it generates a reminder LISP record for an attendee whose bid or validation was not received within a certain period of time, this record is put into the `shared global message queue` as an "agent-reminder". From there, the automaton can process it as each other request. The details of the `time-out handler` can be found in section 2.2.5.

[20] With one exception: If the user commanded a `usr-stop-all`, this request is put at the back to let the agent finish the processing of the pending messages in the queue.

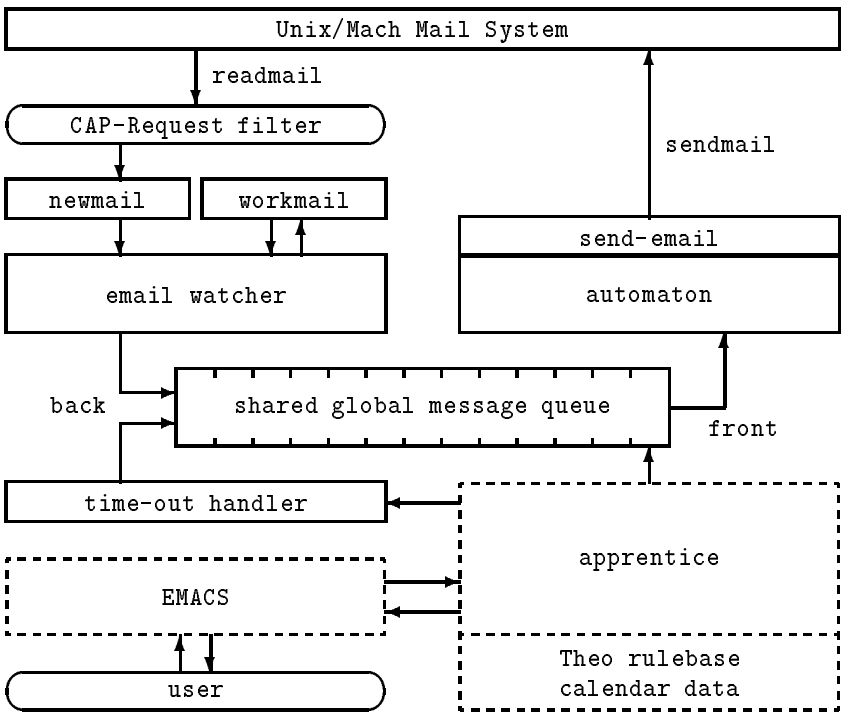


Figure 24: The process structure of CAP II. The dashed boxes indicate components of the original CAP system.

4.2 The Interface between CAP and CAP II

4.2.1 The Communication Interface

A special mechanism had to be implemented to provide access from the CAP II agent's sub-processes back to the EMACS interface. This is always necessary when the agent wants to "ask" the user something if it cannot solve some problem automatically or if it just "wants" to inform its owner. Basically, EMACS has only poor facilities to let one program such an interface.

Our solution looks as follows: Within the EMACS-LISP interface process, a so-called `emacs-listener` sub-process periodically reads from a named pipe (under Sun/OS; under MACH, communication via sockets is needed). It expects to read an EL-LISP expression which then is evaluated immediately. The return value of the evaluation is converted to a string and written back to another named pipe where it can be read from by the external LISP process that makes up CAP II. (In the case of errors the EL-LISP error message is written to the pipe.)

To allow the external LISP process an undisturbed communication (i.e., an expression is sent

to EL-LISP for evaluation and the *corresponding* return value is be waited for), we provide two semaphor-protected functions

`(protected-emacs-funcall <lisp-expression-as-a-string>)`

`(protected-emacs-prompt <prompt-string>)`

The first function can be used to "inform" the CAP interface about something, the second one is necessary to *ask* the user for something (note that `protected-emacs-prompt` constructs an appropriate EL-LISP expression and then uses `protected-emacs-funcall`).

4.2.2 Mapping CAP II Events to CAP Events (and vice versa)

A couple of functions had to be provided to map CAP II events (i.e., LISP records) to CAP events (i.e., records stored in the THEO database), and vice versa. The function

`construct-and-enqueue-mess-record`

is used in CAP after an `add-event`, `move-event`, or del-event was performed and the user chose the option to call the CAP II negotiation package. The function constructs a CAP II message (type usr-proposal) from the THEO data and enqueues it in the shared global message queue where the CAP II automaton can read and process it.

The following functions are used from the CAP II agent to transmit information to CAP, change something in the calendar display, or change something inside the THEO database.

- `create-named-request`: A new event is created and entered into THEO after an attendee's agent received a proposal of a host the first time.
- `delete-event-in-display`: An event is deleted from the calendar display (but not in THEO).
- `delete-event-in-theo`: An event is deleted in THEO.
- `confirm-event-in-display`: If the `yes-no` parameter of this function is `true`, the question mark in front of the displayed event (if there is one) is deleted; else, a question mark will be displayed.
- `confirm-event-in-theo`: The confirmation slot of an event in THEO is set to the value of the `yes-no` parameter of this function.
- `move-event-in-display`: The information about an event is moved to another position in the calendar display.
- `update-event-in-display`: The information about date, time, and duration of a meeting is updated in the calendar display.
- `update-event-in-theo`: The information about date, time, duration, location, and attendees of a meeting is updated in the THEO database.

- `notify-update-reason`: This function is necessary to tell THEO whether an update before was caused by the user or the agent. The information influences the way learning takes place.
- `display-attractively`: The event is displayed such that the user's attention is "attracted for sure".

4.3 User Interaction with CAP II

When a user works with CAP and CAP II, it is an *option* to start the autonomous CAP II negotiation facilities. The following, *user-initiated* interactions are possible.

- At the end of each `add-event`, `move-event` or `delete-event` call in CAP an additional prompt `"Do you want to negotiate that event, yes/no [no]:"` with default value `no` is displayed. Only if this prompt is answered with `yes`, the CAP II mapping function `construct-and-enqueue-mess-record` is called that writes the appropriate CAP II record into the shared global message queue (see also 4.2.2).
- When the user had not started the negotiation at the `add-event` time, s/he can do it later by selecting the service `negotiate-meeting` (EMACS key `"N"`; only possible for events of type meeting).
- The user can ask CAP II to display the status of a negotiation at any time by using the service `view-negotiation` (EMACS key `"V"`).

Agent-initiated interaction can happen at any time in the form of

- changed entries in the calendar display,
- information displayed in the minibuffer line, or
- prompts displayed in the minibuffer line.

Only in the last case the user is expected to react, i.e., to answer the prompt. The agent-initiated interactions of CAP II are interfaced to CAP via the functions described in 4.2.2.

5 Conclusion

We have presented an autonomous agent that probably will be part of every future *office software system*, the calendar manager CAP II. CAP II is an experiment to show the usefulness of office tools that *adapt to the user's preferences* and can *act on his behalf*. An important feature of CAP II agents – and of all other office agents too, we predict – is that they cannot only communicate amongst themselves, but also with human workers who don't have that particular program. This is due to the readable, semi-structured message format that, under the *good-will assumption* (see above), does not require any expensive natural language processing.

Currently, about half a dozen people are using CAP at CMU. In this initial field test we have found that CAP's rules provide satisfactory advice after 2 – 3 months (assuming the frequency of meetings of a professor at CMU). CAP II is – right now – used by only one person (the first author), treating all other members of the CAP team as non-CAP users when sending invitations to group meetings etc. The release of CAP II within the CAP group will happen in the next weeks.

6 Future Work

In this chapter we will, first, describe the goals and projects that have already started (or are just about to start) in the CAP group. They are not only aimed to enhance the CAP II agents for calendar administration, but will also head towards further applications like a room apprentice for meeting room reservation. On the other hand, most of the new projects have a more general character, i.e. they will explore fundamental questions of how to embed novel learning techniques, how to integrate external knowledge sources, or how to involve the user in such a way that the agent can "learn" as much as possible. In section 6.2 of this chapter we will present the overall view - also called "our vision" - of the new projects: The *Software Secretary* of the future.

6.1 Extensions of CAP II and Ongoing Projects

In the following, we will present our ongoing work in extending CAP II and developing new concepts for learning apprentice systems in the office automation domain.

Counter Proposals in CAP II
As briefly mentioned in chapter 3.2.1, the implementation of *counter proposals* is on the way. The email parser has already been extended to cope with that type of bid. As the first step we will only allow *one* counter proposal as a bid, i.e. the "send the n best date/time proposals" bidding strategy of Sen and Durfee (with n = 1; see [18]). With this feature CAP II will cover the *full scope* of interactions for the purpose of meeting negotiations as it is performed by human beings.

One additional module, however, has to be developed that is not yet part of CAP II: A reactive scheduler. This scheduler has to solve the problem of finding all *technically possible*

time slots for the original and all counter proposals. It must be reactive, because *move requests* - that also may contain counter proposals - can now be received at *any* time. For the selection of the *(m) best* technical solutions we plan to use the rules learned by the CAP II agent from its owner's preferences.

EFORMS for CAP II

EFORMS (Electronic FORMS) is a means to give email messages a structure and let people respond to requests just by checking some fields or entering some, small values. If EFORMS were exchanged like email today, people in offices would be able to do their jobs much more efficiently. Furthermore, learning agents could derive more information from those structures than from plain email texts and would be able to handle larger portions of the necessary processing automatically than today. As a last advantage, EFORMS could provide substantial support for a natural language interface, since their structure constrains the interpretation of user speech drastically.

We plan to implement an EFORM-based interface to CAP II to provide further support for the involvement of non-CAP users. If possible, at least within the local CMU networks, we have a kind of "remote pop-up mechanism" in mind (REFORMS = remote EFORMS). It takes a LISP-record containing the meeting information, pops it up on an attendee's screen, lets the user fill out and check open fields and gives help where necessary. If this works satisfactorily, no more email exchange is necessary. If not and for "really" remote people outside CMU, we will try to apply concepts like *Computational Mail* [3] or *Active Mail* [8].

A second experiment shall be the connection of EFORMS to the original CAP apprentice, i.e without the agent. The interesting question is whether human beings, who possess the comfortable EFORM support, *still want* to have an agent working in the background. They would keep *full personal control* without the agent, but - that's the drawback - loose assistance in the case of absense etc.

Usage of External Knowledge Sources

Underlying CAP II's ability to schedule meetings and conduct negotiations is a considerable body of knowledge about people and groups at CMU. Since most of this knowledge has been entered manually by the user in the process of scheduling meetings, its scope is limited to whatever information about a person is known by the user and convenient to type in. However, not only is some of this information readily available in the CMU computing environment (e.g., via "finger"), but significantly more useful information might also be retrieved and used by a "software secretary." In other words, if we can extend the system to make modestly intelligent use of the information sources accessible to it in a networked environment, its autonomy, and probably its value to the user, will be enhanced.

We have begun work on such an extension. As a domain for knowledge acquisition, a computer network is subject to considerable noise and variability. Our initial efforts in this endeavor, therefore, will consists in the design of an extension to CAP II's knowledge base, under Theo, which will equip it with an ability to reason about information sources accessible to it and the information it retrieves from these sources. Eventually, we hope to add a DBL module to this subsystem through which the agent will be able to integrate new information

sources by appealing to the user for help.

System Design, User Interface, Machine Learning, and the Handling of Irregularities

"Conventional" or "classic" machine learning does not seem to be doing particularly well in the world of apprentices and agents. We believe that one of the reasons lies in the relationship between system design, user interface, and machine learning.

In the development of an agent, the system design has to come first. Unfortunately, there clearly exist trade-offs between the UI (user interface) and the application of ML (machine learning) which have to be balanced by the design. One example for design decisions based on such tradeoffs is the tuning of parameters within the parameter space in order to support a certain adaptability of the system for the user. For that purpose, a good system design needs to:

- Restrict the dimension of the parameter space significantly.
- Give good and intuitive directions in the parameter space along which the parameters should be tuned.
- Provide easy and automatic ways to tune parameters in the above directions.

In the last item, the term "easy ways" corresponds to UI and "automatic ways" corresponds to ML.

For a good UI we need a "smart" and "simple" (in terms of the user interface) language and/or GUI (graphical user interface). They should be smart enough to handle *irregularities* and *exceptions* to some extent. With such a language or GUI, the user does not need to wait until the system learns the rules for setting up schedules. It should not cost the user very much if the irregularities could be acquired as simple as telling her/his secretary the rules. We are afraid that users do not even start to use a system without such a language or GUI in the first place.

Take the scheduling task, for example. The regular meetings are usually the large part of all meetings. With the help of a suitable language and/or GUI, users should not have much trouble to give rules, not only for regular meetings but also for some irregularities or exceptions.

The next question is, where does machine learning fit to the task of acquiring irregularities? "Conventional" or "classic" machine learning methods usually provide automatic parameter tuning in the predetermined directions in the parameter space from statistical observation of data. But there are many adversaries in the above setting.

Because of their irregularities, there are only a few examples. That makes it difficult for any system to observe them statistically.

Irregular data is often irregular *in its construction*. Some features might be missing. For example, irregular meetings often involve strangers, and information about them is sometimes simply unavailable. Also giving reasons for irregular meetings in a machine understandable way might be difficult.

Even if there exist some rules which apply to irregular data, users are often unaware about these rules or the features used in those rules.[21] Sometimes features necessary for expressing these rules are missing in the set of features given for ML at first.

We believe that one of the possibilities for ML to be successfully applied to this kind of tasks is to *change the parameter space itself* dynamically. Not dealing with a fixed set of features, the system should search for a new set of possible features adaptively.

RAP: Learning the Automaton for a Room APprentice
In the RAP project, a learning CSCW[22] agent for reserving meeting rooms shall be implemented and explored [2]. It shall watch a secretary when sending email messages to the responsible persons at CMU and extract (i.e. learn) the "protocol" of that task incrementally. Imagine, for example, that the secretary S1 mails to the responsible "room administration person" P1 in the CS main office to reservate a room for a certain day. S/he receives the answer that no room is available on that day. Then s/he tries the same with the "room administrator" of the robotics department, and so on. The RAP agent will watch those email "dialogs" and shall learn the following (see also the next issue in this section):

- A little finite state machine (FSM) that decides what to do after receipt of which message (or command).
- The "vocabulary" for the FSM, i.e. which keyword or keyphrase decides the next transition to perform.[23]
- Which messages should be generated at which state.
- Which messages should be presented to the user, because they require complicated treatment.
- A little database that contains information about rooms (or how to access it in CMU databases), about persons that can be asked, and maybe also about preferences of requesting people, priorities of "meeting types" etc.

AAP = CAP + RAP: An Agenda APprentice for CMU Visitors
If RAP is successful we have in mind to combine it with CAP in order to support a really bothersome task: The setup of an *agenda* for visitors of CMU.[24] For that job, the visitor's host has to schedule meetings with interested people as well as to reserve a room for a talk if the visitor is invited to give one. Unfortunately, as the own experience of some of us has shown, those agendas tend to be "open" till the last minute. Program support, therefore, would be heavily appreciated.

[21] If the user is aware about them, then there are good chances that s/he has already given the system those rules by the language or GUI. But if the users are lazy, it is not always the case.

[22] Computer-Supported Cooperative Work.

[23] See chapter 3.1 for the discussion of how close to a "natural language" interface we want to go.

[24] All people who ever did that job will probably be immediate customers of such an apprentice, we guess.

Programming by Demonstration and Dialog-Based Learning Techniques for Interface Agents

As already pointed out in chapter 1.3, apprentices are a kind of (implicit or integrated) programming by demonstration system. The extent, to which *learning* can be successfully applied in such systems is not yet deeply explored. We believe that the "classic" Machine Learning techniques are not suitable enough if applied *without* substantial support from the user. First, the number of available training examples grows slowly. Clever PAC-like[25] learning algorithms [21], even when polynomially bounded [17], don't seem to be the right way. EBL[26] [5] requires a lot of modeling, but a human end-user, i.e. all the actions he will perhaps perform, cannot be modeled "that easily." The same holds for action models if they shall be learned by neural networks. On the other side, neural networks as well as decision tree learning has been proved to be a substantial part of our CAP II agent to acquire its owner's preferences in order to give better advice.

Therefore, we propose a technique that can co-exist with all other learning approaches, but will make (more) use of the person sitting in front of the interface agent: *Dialog-based learning* or *DBL* [2]. That means for CAP II or RAP, the agent "asks" its owner if it cannot "understand" a certain email or cannot decide which action shall be taken next. For example, if the RAP agent searches for keywords like `free` or `occupied` in an answer message to a reservation request, it can present a message without those words to the user and "ask" what is the meaning-carrying word/phrase. Suppose, the user gives the answer `unavailable` because the message announced that all rooms were blocked on that day. Then, the agent could ask, whether `unavailable` is a *new* keyword or a *synonym* for one it already knows. If the user says "it is a synonym," namely to `occupied`, the agent will enhance its internal thesaurus.[27] After that it can process the *next* message of that kind autonomously as it had done in the case of `occupied`. If the user answers "it is *new*," the agent will add a new state to its FSM and define `unavailable` as the necessary keyword to enter that state. In the RAP project (see above) we want to explore that approach on a practical problem for the first time.

DBL may, perhaps, better be called knowledge acquisition than learning.[28] However, according to the definition by H. Simon [19], an agent that uses DBL learns, because it will perform the same task *more efficiently and more effectively* the next time after it has acquired new knowledge from the user. *More efficiently* - in the example described above -, because a long search and/or the user interaction time is saved. *More effectively* for the owner, because the agent can process the same email now autonomously and does not require the user's time or attention.

Altogether and independent from any terminology quarrels, we believe that DBL-like mechanisms are a promising way to enhance interface agents substantially. "Classical" Machine Learning algorithms are too limited for being the only learning mechanism in agent systems.

[25] PAC = Probably Approximately Correct.

[26] EBL = Explanation-Based Learning.

[27] Note, that a first application of that method is already part of CAP II. Alias lists for attendees' names and email addresses are learned by asking the user, if messages are received where the sender cannot be determined by the agent. This happens mainly when mailer demons add some node information about particular machines to the from-field [2].

[28] Michalski (see [14], p. 10), nevertheless, calls the knowledge acquisition aspect of learning to be the essence of most learning acts.

Furthermore, it would be a big mistake - in our opinion - to ignore the human end-user who *is present*, i.e. sits in front of its agent-containing tool, at least during the usual work time. Dialog-based learning, therefore, will be a key issue in the further research within the apprentice group at CMU.

6.2 Our Vision: The Software Secretary

Our ultimate vision is an office automation system that one might call *a software secretary*: a software agent that provides assistance in managing work-related activities, in much the same way as a human secretary. The concept of software secretaries seems clearly attractive, since such software could make available to everyone within an organization the kind of secretarial assistance that is presently available only to a small fraction of workers.

While the technology for developing such knowledge-based systems is sufficiently advanced to pursue our vision of software secretaries, such systems are not practical using today's software development methods. This is because tasks such as calendar management require knowledge that is highly specific to each individual user (e.g., individual meeting preferences, job requirements), but might also be very specific within a paricular organization (e.g., constraints on meeting times and locations). Furthermore, such knowledge changes significantly over time as the organization and individual grow and evolve. Therefore, one has to study cooperative learning among multiple agents in pretty much the same way as the study of cooperative problem solving already happens.

It is also clear that *automated* learning procedures alone will not produce the results we expect. Hence, an important part of our future work will be the experimentation with dialog-based learning (DBL) in order to benefit from the users of software secretaries. They can make their intentions accessible such that the agents need not blindly search in some hypothesis space made up by the observed data. Instead, derivations are focussed on the properties of the user interaction that really are essential for a learning step. In a certain sense one can even say that the users are able to "program" their office software themselves, i.e., they provide it with their individual rules and parameters which adapt the software optimally to their personal needs [1].

We know that our goal of a software secretary is rather ambitious. However, we believe that software packages of this kind are the only way to help computer system *users* of tomorrow in managing the amount and complexity of their daily work in an always faster changing world of high-tech equipment sucessfully.

Acknowledgements

The authors gratefully acknowledge various fruitful discussions and valuable comments by John and Liam McDermott and David Zabowski. Dave also assited a lot in coupling CAP II to CAP by writing the `emacs-listener` as well as some of the mapping functions. We also want to thank Sybille Rosenmüller and Gisbert Lawitzky who always handle the necessary "remote bureaucracy jobs" in Munich quickly and reliably.

References

[1] S. Bocionek and T.M. Mitchell. Office Automation Systems that are "Programmed" by their Users. In *Proceedings der 23. Jahrestagung der Gesellschaft für Informatik, GI '93*, Dresden, Germany, September 1993. Springer-Verlag.

[2] S. Bocionek and M. Sassin. Dialog-Based Learning (DBL) for Adaptive Interface Agents and Programming-by-Demonstration Systems. Technical Report CMU-CS-93-175, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, July 1993.

[3] N.S. Borenstein. Computational mail as network infrastructure for computer-supported cooperative work. In J. Turner and R. Kraut, editors, *CSCW '92 (Sharing Perspectives)*, pages 67 – 74, Toronta, Canada, November 1992. ACM Press.

[4] J.C. Brustoloni. Autonomous agents: Characterization and requirements. Technical Report CMU-CS-91-204, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, November 1991.

[5] G.F. DeJong and R.J. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145 – 176, April 1986.

[6] L. Dent, J. Boticario, J. McDermott, T.M. Mitchell, and D. Zabowski. A Personal Learning Apprentice. In *National Conference on Artificial Intelligence (AAAI-92)*, August 1992.

[7] E. Egger and I. Wagner. Time-management - a case for CSCW. In J. Turner and R. Kraut, editors, *CSCW '92 (Sharing Perspectives)*, pages 249 – 256, Toronta, Canada, November 1992. ACM Press.

[8] Y. Goldberg, M. Safran, and E. Shapiro. Active mail - a framework for implementing groupware. In J. Turner and R. Kraut, editors, *CSCW '92 (Sharing Perspectives)*, pages 75 – 83, Toronta, Canada, November 1992. ACM Press.

[9] J. Jourdan, L. Dent, J. McDermott, T.M. Mitchell, and D. Zabowski. Interfaces that learn: A learning apprentice for calendar management. Technical Report CMU-CS-91-135, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, April 1991.

[10] R. Kozierok and P. Maes. A learning interface agent for scheduling meetings. In *Int. Workshop on Intelligent User Interfaces*, Orlando, Fl, January 1993. ACM-SIGCHI, ACM Press.

[11] T. Kreifelts. Coordination of distributed work: From office procedures to customizable activities. In W. Brauer and D. Hernandez, editors, *Proceedings of 4th International GI-Kongreß Wissensbasierte Systeme (Knowledge-Based Systems)*, pages 148 – 159, Munich, October 1991. GI, Springer-Verlag.

[12] A. Lux. A Multi-Agent Approach towards Group Scheduling. Research Report RR-92-41, Deutsches Forschungsinstitut für Künstliche Intelligenz DFKI, Kaiserslautern, August 1992.

[13] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The ActionWorkflow approach to workflow management technology. In J. Turner and R. Kraut, editors, *CSCW '92 (Sharing Perspectives)*, pages 281 – 288, Toronta, Canada, November 1992. ACM Press.

[14] R.S. Michalski. Understanding the nature of learning: Issues and research directions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning II, An Artificial Intelligence Approach*, chapter 1, pages 3 – 25. Morgan Kaufman Publishers, Inc., Los Altos, CA, 1986.

[15] T.M. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, and J.C. Schlimmer. THEO: A framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence*, chapter 12, pages 323 – 355. Erlbaum, 1991.

[16] B.A. Myers. Demonstrational interfaces: A step beyond direct manipulation. Technical Report CMU-CS-90-162, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, August 1990.

[17] R.E. Schapire. The design and analysis of efficient learning algorithms. Technical Report MIT/LCS/TR-493, MIT, Laboratory for Computer Science, Cambridge, MA, February 1991.

[18] S. Sen and E.H. Durfee. A formal study of distributed meeting scheduling: Preliminary results. In *Conf. on Organizational Computing Systems*, pages 55 – 68. ACM, November 1991.

[19] H. Simon. Why should machines learn? In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 2, pages 25 – 37. Tioga Publishing Comp., Palo Alto, CA, 1983.

[20] J. Turner and R. Kraut. *Sharing Perspectives: Proceedings of the CSCW '92.* ACM Press, November 1992.

[21] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134 – 1142, 1984.

[22] M. Woitass. Coordination of Intelligent Office Agents – Applied to Meeting Scheduling. In S. Gibbs and A.A. Verrijn-Stuart, editors, *Multi-User Interfaces and Applications*, pages 371 – 387, Heraklion, Greece, September 1990. IFIP, Elsevier Science Publishers B.V.

[23] M. Woitass. *Koordination in strukturierten Konversationen*, volume 190 of *GMD-Berichte*. Oldenbourg Verlag, 1991.