Dialog-Based Learning (DBL) for Adaptive Interface Agents and Programming-by-Demonstration Systems

Siegfried Bocionek Michael Sassin[†]

July 1993
CMU-CS-93-175

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]AI Laboratory, Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139

This research was performed during a one year stay as Visiting Scientists at CMU and MIT, respectively, funded by Siemens AG, Corporate Research and Development, Munich, Germany.

Keywords: Artificial Intelligence, dialog-based learning, programming by demonstration, interface agents, office software, graphics editor, Software Secretary

Abstract

Many users of workstation and PC tools often have to perform the same task again and again. For example, a secretary might have to send out a dozen email messages until she finds a free meeting room. Or someone preparing business charts has to draw many special tables with shadowing bars around. Unfortunately, today's macro facilities of such tools do not support the end user enough in constructing the required automation functions.

In this report we propose a mechanism, called dialog-based learning (DBL), that shall provide the user of software tools exactly with a mechanism to teach new functions or to give hints or additional information to a program on how to perform a task better. Two applications will be considered: The first one is our experimental system RAP, a room reservation apprentice that will eventually overtake a secretary's task to search for a free meeting or lecture room. RAP analyzes the outgoing and incoming email and constructs a finite state machine that can repeat the task of asking all room administrators until a free room is found. The key of RAP's learning is to ask the user for unknown message types (e.g., request, positive answer, etc.) and keyphrases (e.g., "need a room") and to collect them in a thesaurus.

Our second application is a demonstrational graphics editor that allows the user to teach it new functions by giving a few examples. The graphics editor will sometimes ask the user for explanations by showing him a list of its geometrical hypotheses for a certain situation, e.g., "line l1 was doubled" or line "l1 touches line l2 in the middle" (the second hypothesis serves as a strong indication that "touching" is the intended property). By clicking at one of them the user *tells* the graphics editor his intention and helps it to construct a function and a menu item for a new function.

Involving the user through dialogs is an easy way to support the heuristics for function induction from examples when the hypothesis space grows. Our claim is that – with DBL – a Programming by Demonstration system can synthesize new functions faster, that it can synthesize more complex functions than without DBL, and that the resulting functions will meet the user's intentions better (because he or she took part in their derivation). Therefore one can see DBL as a kind of simple inference mechanism, but with a powerful outcome.

Contents

1 Introduction **3**
- 1.1 Why Should One Use Dialog-Based Learning? . . . 3
- 1.2 Issues Adressed in this Report . . . 4
- 1.3 Organization of the Report . . . 6

2 Background **6**
- 2.1 Definitions . . . 6
- 2.2 What is Dialog-Based Learning? . . . 7
- 2.3 Related Work . . . 8

3 DBL for Knowledge Acquisition **9**
- 3.1 Acquisition of the Knowledge Base in CAP . . . 10
- 3.2 Proposal on how to Support Rule Learning in CAP . . . 10
- 3.3 Acquisition of Alias Lists in CAP II . . . 11

4 Learning the RAP Automaton with DBL support **12**
- 4.1 The Restrictions of RAP . . . 13
- 4.2 The Initial Domain Knowledge of RAP . . . 14
- 4.3 Learning the Vocabulary for RAP . . . 16
- 4.4 Learning the Finite State Machine for RAP . . . 17
- 4.5 Discussion . . . 22

5 DBL as a Support for Programming by Demonstration **25**
- 5.1 The PbD Concept for the Graphics Editor . . . 25
- 5.2 Problems . . . 26
- 5.3 DBL as a Solution to Handle Incomplete Domain Theories . . . 29
- 5.4 Embedding of DBL into a Graphics Editor . . . 30
  - 5.4.1 Interpretation of a Single Action . . . 30
  - 5.4.2 Interpretation of Action Sequences . . . 36
  - 5.4.3 Dialogs during Generalization . . . 38
- 5.5 Learning New Property and Relation Functions . . . 39
- 5.6 Discussion . . . 44

6 Conclusion **45**

7 Future Work **46**

List of Figures

1 A meeting proposal generated by a CAP II agent 13
2 The meta-template for email messages in RAP 14
3 A request for a lecture room as generated by a RAP agent 16
4 PbD concept for a graphics editor 26
5 Ambiguity of explanations 27
6 Rotation of objects around a fixed point p 28
7 Panel with six explanations of a change-line action 30
8 A graphics situation 31
9 Properties and relations of line l_1 32
10 Manipulation with the help of property and relation forms 33
11 Example of a metacalculator in a PbD system 34
12 Move a line to the meeting point of two other lines 35
13 Example of a history board 36
14 Two nearby-parallel lines 38
15 A first negative example for nearby-parallel lines 39
16 A second, more complex negative example 40
17 The interaction with the metacalculator 41
18 The first positive example for nearby-parallel lines 42

List of Tables

1 RAP's empty FSM 18
2 A little part of RAP's FSM 18
3 The algorithm of RAP's learning FSM (example in brackets) 20

1 Introduction

Dialog-Based Learning (DBL) is a method to support adaptive programs by means of user interaction. The system improves continuously by performing question and answer dialogs with the user. The two major examples to be discussed in this report are the task of reserving rooms via email and the task of constructing new functions for a graphics editor by showing it a few examples.

The first task shall demonstrate how an apprentice system can acquire workflow protocols by watching a user who sends and receives email messages, by analyzing the messages, and by asking the user for keyphrases etc. in them. Our experimantal system is the room reservation apprentice RAP which builds up a thesaurus of message types with keyphrases and synonyms, as well as the finite state machine (FSM) that performes the automatic search for a room.

Our second task, the teachable graphics editor, shall demonstrate how the interaction with the user through dialogs facilitates the problem of inducing functions from examples. By selecting the right hypotheses in a form-like window, the user informs the graphics editor about his intentions. That helps the editor to shrink the search space, e.g., reduce the number of variables to be considered when constructing the precondition for applying a function.

Another learning apprentice system will also be referred to. It is the calendar apprentice CAP [8, 11] which acquires its user's preferences in setting up meetings and gives advice (e.g., for the duration and location of a meeting) when entering a new meeting in the online calendar. Besides the really used CAP program,[1] the experimental system CAP II [1] will be mentioned, too. It extends CAP by a special interface that automates meeting negotiation via email between CAP II agents and also humans without that program. However, the negotiation protocol is "hard-wired" and cannot be learned as in RAP.

1.1 Why Should One Use Dialog-Based Learning?

There are three major reasons *why* DBL should be used in systems with automated learning capabilities, especially in *interactive systems* like the email environment or the graphics editor discussed in this report.

- **Number of training examples grows slowly**
  In the case of RAP one might have one reservation a week or even less. And in the case of the graphics editor, nobody wants to provide more than a handful examples in order to teach a new function. Therefore, automated learning is difficult. PAC-like learning algorithms [37], even when polynomially bounded [33], need very many examples. ID3-like algorithms need less, but more than a user of office software provides in a reasonable time. EBL [7] needs only one example, but requires a large amount of modeling. However, modeling the end-user, his possible goals, intentions, and behaviors is almost impossible.

- **Many differences and exceptions**
  Especially the office automation domain has to cope with many, often only *slightly* different tasks. What works correctly in office A might require additional treatment in office

[1] About 6 - 10 people use CAP, partly for about 2 years.

B. In the case of RAP, in one organization one might have to ask especially installed room administrators; in others, one might have to negotiate with "room owners" directly. Therefore, the user should – via dialog-oriented mechanisms – get the possibility to easily *customize* the tool according to his environment.

- **The user is present**
  Using office software, the user is available most of the time and can be *asked* in case of a problem like a email message that cannot be "understood" by the system. The user can (and should) also be asked by the system during generalization steps. In the case of the graphics editor, where the user is necessarily present during demonstration, it might help the system a lot to know that the intention in a specific drawing (step) was to "double the length of a line" although it crosses another line after that which was not touched before. The second line might then not be included in the induction steps (according to the used heuristics). Supporting the generalization by asking the user can ease that job heavily in the case of many possible explanations, a large hypotheses space, or at any other point of decision.

In addition to the *reasons* that recommend the usage of DBL in interactive systems, there is a "philosophical" argument for it as well. This argument, based on some of Heidegger's ideas, is nicely discussed in H. Dreyfus' new edition of *What Computers Still Can't Do* [9]. It says that there is no need for too much modeling of the world in a computer program, because a human being also does not maintain models most of the time. Instead, one should just use the "environment" in some "reactive" manner (*... the worlds is its own best representation*; [9], p. *xxxi*[2]). Supporting our last reason, that the user is mostly near his interface agent, we interpret Dreyfus' words as "Make use of the user; Ask him". Or, in our terminology, model as little as necessary, but provide DBL support for the agent.

1.2 Issues Adressed in this Report

Extensibility
Extensibility of software systems is the major issue addressed in this report. Extensions shall be performed in a demonstrational way and shall be supported by DBL. We consider various kinds of extensions (or learning tasks, see below):

- Acquisition of fact knowledge to build up internal databases. Having that knowledge the system will disturb the user less than before. One example is the learning of a thesaurus of keyphrases in our room reservation apprentice RAP. The user will never be asked for each of the entries again.
- Acquisition of user preferences in order to provide better system support. The example is the learning of rules in CAP where to schedule a meeting and how long.

[2]The quotation is, originally, from D. Chapman [4], p. 20.

- Acquisition of a thesaurus of expressions the user exchanges when working together with other people through a computer network. Our example are the keyphrases in the email messages that are sent around when trying to reserve a room with RAP.
- Acquisition of procedural knowledge that describes the user's strategy to work with others (i.e. workflow concepts). This knowledge can be used by the system to overtake more and more jobs. The example is the incremental construction of the FSM in RAP that eventually overtakes the complete search for a room.
- Acquisition of new functions by demonstrating their functionality with the existing ones. The mechanism can be used for user-customization as well as for a general, boot-strapping-like software development method. An example in the graphics editor domain is the acquistion of a function that extends a closed polygon to a 3D polyhedron as described in [31]. That function can be shown just on the basis of a "draw-line" function that is used to draw all the parallel lines at the corners of the polygon and finally connect the endpoints.
- Acquisition of new interpretation functions that extend the domain theory of a PbD system in order to make it more powerful. In this report we will show the acquisition of the concept "nearby-parallel-lines". Having this function extends the set of available interpretation functions of the graphics editor, i.e., its domain theory.

We want to show that in all those cases *dialogs* are a key component which help facilitate the task massively.

Promising Example Applications
We have chosen applications which are of practical significance, and are also suitable enough to show the principles of our approach. The domains are *learning apprentices for office automation tasks* and an *extensible graphics editor*. As partly already mentioned above we will consider the

- Acquisition of facts and user preferences in the calendar systems CAP and CAP II.
- Acquisition of a thesaurus and a finite state machine for the room apprentice RAP.
- Acquisition of new functionality and additional domain theory in a graphics editor with programming-by-demonstration interface.

Generality
An important issue is generality, i.e. to show that our approach is not limited to the example applications. This point will mainly be discussed in the last sections of each of the major chapters 4 and 5.

Scalability
In chapter 5 we sketch out possible methods in order to make the DBL approach also applicable in case of large numbers of possible explanations in the domain of a graphics editor. The methods are general enough to be also applicable in other domains. Additional ideas for the graphics editor and user interface domain can be found in [13]. In the apprentice systems case we have not seen work yet that addresses the problem.

1.3 Organization of the Report

The remainder of this paper is organized as follows. Although we have already used terms like apprentice or agent, we will first give some short definitions to make exact what we understand by those words. In section 2.2 we will present our view *why* we consider DBL to be a kind of learning and *where* it fits the actual classification of ML as given in [21], [22] and [12]. After that, we possess the vocabulary to discuss related work in section 2.3.

The chapters 3, 4 and 5 contain the example applications CAP, CAP II, and RAP from the learning apprentices domain, and the extensible graphics editor, respectively.

In chapter 6 we will conclude our results followed by a list of planned research topics, applied as well as basic ones, in chapter 7.

2 Background

2.1 Definitions

A *Programming-by-Demonstration* or *PbD* system allows its user to construct programs on an abstraction level highly above traditional programming languages. The major application area is graphics editors [18] and graphical user interfaces [28], where the user interacts with mouse, menus and icons as he is used to. New procedures, e.g. the extension of two-dimensional polygons to three-dimensional polyhedra, are "programmed" by just moving the graphical symbols around and not by typing any line of code. The "programming" takes place in a special demonstrational mode.

We speak of an *embedded Programming-by-Demonstration system*, if there is no extra demonstrational mode to derive new procedures. They are acquired by *watching* the user's actions and synthesizing the traces to code. One successful application are *apprentice systems* that give advice to the user.[3] They acquire, for example, new rules for better advice, if the user overwrites values that they had recommended (see [11] and [8] for a description of the calendar apprentice CAP).

An apprentice system might be a kind of "passive", advice-giving tool. It might also be an active *agent*, if it *acts purposefully* in the real world in an *asynchronous* or *reactive* manner. Furthermore, an agent must have some *decision capability*, i.e. it is allowed to select processing steps without disturbing the user. It may even act on its owner's behalf. A fourth required property for an agent is *long-term existence* (see the discussion in [1]).

An *interface agent* is an agent connected to an interactive tool, e.g. the calendar program CAP.[4]

[3] According to the definition in [1], apprentice systems are knowledge-based, interactive programs that give advice and assist the user in daily tasks, e.g. in the office work domain or for VLSI design tasks etc.

[4] An agent which does not fit in that class is, for example, a path planner in an intelligent workcell that can be invoked reactively by some cell controller. It performs its planning job asynchronously while the controller continues to supervise the cell components. There is no user interface to the path planner in such an application.

An interface agent is *adaptive*, if its behavior changes according to its users needs and preferences, and, hopefully, *improves* over time. We use adaptive as a synonym for *learning*, because the definition is consistent to that of H. Simon in [35].

There may be many different ML algorithms embedded in learning interface agents. Our suggestion is to include Dialog-Based Learning (DBL) as one major technique as well. The next chapter is intended to clarify what DBL is and how it fits the ML terminology.

2.2 What is Dialog-Based Learning?

Trying to give a one-sentence definition we say

> A program contains Dialog-Based Learning (DBL) if it asks the user – in an asynchronous or also synchronous way – for some information and uses this information to improve its own functionality or efficiency.

As stated previously, DBL *is* learning. An agent which has got some new information via DBL, will perform the same task *more efficiently and more effectively* the next time (compare to [35], p. 28 and [1], chapter 7.1). Some people may prefer to call DBL *knowledge acquisition*, especially in the simple, fact-gathering case. According to R. Schank and A. Kass, however, simple knowledge acquisition can "legally" be called learning (see [32], p. 34). Moreover, following R. Michalski in [20], p. 10, the "...knowledge acquisition aspect of learning seems to be the essence of most learning acts".

An interesting addition of R. Michalski to H. Simon's definition is that the central aspect of learning is "...constructing a representation of some reality ..." (see [20], p. 11). We will refer to that point later in the discussion of our DBL applications.

DBL can (mostly) be subsumed under *symbolical empirical learning* [23]. DBL is *symbolical*, because the new gathered information is represented by symbols with fixed syntax and semantics: E.g., CAP learns rules while PbD represents new knowledge as function schemata or procedures. DBL is *empirical*, since it is not based on a full, previously known theory as in EBL [24], [7]. Additionally, it is partially based on observations of events which have to be interpreted with the help of a user. In fact, DBL should be empirical. Otherwise DBL had all the domain-modeling problems in full generality (see chapter 1), since the pre-programmed domain description is in general not large enough in order to enable interface agents deducing all necessary knowledge.

DBL can also be seen as *synthetic learning* [23], because it helps to develop a new, larger theory based on the old, week domain theory. In our opinion DBL and EBL provide a powerful combination (see also [1], chap. 7.1).

According to the classifications of learning strategies in [3] and [20], DBL may *mainly* - not exclusively - appear as:

- *Rote learning*, if facts are just collected and memorized.
- *Learning from instruction* or *learning by being told*, if transformations (e.g. to rules, procedures, semantic nets etc.) are necessary to make the new knowledge useful for an agent.

- *Learning from examples*; that is just a variant of the latter.
- *Learning from question and answer pairs*; that is one possibility to define examples.

These learning strategies also define, roughly, the research areas the are most closely related to DBL. We will discuss some results in the next chapter.

2.3 Related Work

A first example for a concept that uses DBL for learning is C. Sammut's and R. Banerij's MARVIN [30] which learns new concepts in the Arch domain. It works as follows: First, the trainer presents MARVIN a positive example. Afterwards, MARVIN tries to induce a new concept by matching the example against some basic and other, already acquired concepts, and by generalization. After that, MARVIN constructs an instance of the new concept and *presents* it to the trainer. If he or she answers with "yes", the instance is a correct example of the intended concept, else it is not. To repair wrongly induced concepts, MARVIN asks the trainer for each part of the example arch whether it is an instance of its *assumed* concept. By that method, the wrong assumption is identified and can be corrected.

A predecessor of the MARVIN-like approach was E. Shapiro's learning of new concepts in a Horn-clause environment [34]. His *inductive inference from facts* scheme was also applied to the arch domain, but mainly for the learning and debugging of little Prolog programs (e.g., quick-sort, concatenation, etc.). For debugging, a so-called *contradiction backtracing algorithm* was introduced to detect missing precedence clauses. The algorithm used a kind of DBL when it prompted the user for the truth of all ground atoms in the proof tree after a Prolog program yielded a wrong result.

DBL mechanisms can also be found for the acquisition of new rules in rule-based expert systems. One famous example is TEIRESIAS, the knowledge acquisition component of MYCIN [6]. DBL-supported factual and rule knowledge acquisition was included in ODYSSEUS, the apprenticeship learning program inside HERACLES [40, 39] (a successor of the NEOMYCIN expert system shell). ODYSSEUS contains *failure-driven EBL* to acquire the new knowledge if no HERACLES metarule can be applied to some diagnosis situation. Then ODYSSEUS constructs so-called *clearifying questions* that shall help it to verify a system hypothesis. If the questions cannot be answered internally (through inspection of the factual knowledge) the physicist is asked, e.g., for duration or severeness of a headache.

A big research effort was also the *instructable production system* project [29] at CMU. In systems like KERNEL, a human instructor had to watch the trace and give advice during problem solving via a dialog interface. The systems used a combination of *learning by being told* and *learning by independent exploration.*

In Programming-by-Demonstration (PbD) systems, which are interactive by nature, one can find a variety of DBL techniques. This can be the usage of popped-up accept buttons or the need to inspect the LISP code of a synthesized function and accept, reject or correct it [27]. Other DBL techniques are the checking or filling of answer fields.

Also more implicit forms of dialogs are covered by the DBL concept as used in the graphics editor METAMOUSE [18]. Here, a "supporting line" is used to indicate which objects has to

be manipulated in the next step. The supporting line is moved continuously, until it touches an object that has to be manipulated next. After the manipulation the supporting line is moved to the next object. At each step the relations of objects according to the former or next action in the graphics editor are recorded. By built-in heuristics repetitions (loops) in actions and constraints are recognized and, based on them, next actions are anticipated.

Another PbD application is Lieberman's TINKER [16]. Guided by positive examples and a programmer with less experience TINKER generates a program. TINKER analyses a sequence of actions of an example and generates LISP code to do the same. If during the analysis of the next positive example an inconsistency in the former-generated function is found, TINKER concludes that a condition has to be included in the LISP function in order to support branching. It asks the user to directly fill in the necessary condition in form of a LISP expression.

DBL techniques in *apprentice systems* will be discussed in the chapters 3 and 4. The described applications will be CAP [11] and CAP II [1] as two different variants of a learning calendar system. A second example is the room apprentice RAP.

One major research domain within AI is *understanding*, that means, usually, understanding of natural language [35], [32]. Although H. Dreyfus claims this goal is unreachable [9], corresponding systems have been - and still are being - developed. One of these Knowledge-Learning And Using Systems is *KLAUS* by N. Haas and G. Hendrix [10]. They developed prototypes like NANOKLAUS and later MICROKLAUS that contained *learning by being told* mechanisms for acquiring a "dictionary" of world knowledge that is organized in type hierarchies and semantic nets. The important point was that their KLAUS systems provided interactive, mixed-initiative dialogs. The hard task was (and is) to acquire new concepts *and* the linguistic constructions used to express them (see [10], p. 406).

Further work, related to DBL, can be found in CSCW[5] publications [36]. Especially important for our apprentice and agent applications in the office automation domain is the general model for all kinds of work in organizations as introduced by Medina-Mora, Winograd and Flores, the *ActionWorkflow Loop* [19]. This work, and many other papers that describe calendar and meeting scheduler systems in particular, is discussed in [1].

3 DBL for Knowledge Acquisition

Three kinds of knowledge acquisition by means of DBL shall be presented in this chapter. First, the acquisition of attribute values for the entities that are handled by the calendar apprentice CAP [11]. These values are necessary to induce the rules for advice giving. As the second example we propose a way to increase the confidence factor of the rule outcome in CAP with DBL. The third example is the acquisition of alias lists by the interface agents in CAP II [1]. Without those lists, the agents could not react autonomously on all kinds of incoming email.

[5] Computer-Supported Cooperative Work.

3.1 Acquisition of the Knowledge Base in CAP

The Calendar APprentice 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.[6] The user can then accept the suggestion or enter another value if he prefers.

CAP's ability to give advice improves over time, because it *learns* the user's preferences about how to set up meetings. The learning of new rules for advice giving is based on the data of meetings (time, date, location, duration), but also records about the participants (affiliation, position etc.) and the events themselves (type like seminar, course, meeting etc.). Learning starts each night by activating some Theo routines [25].[7]

DBL comes into play when the descriptions of attendees or events are acquired. Each time CAP recognizes a new name (of a person, a seminar etc.), it asks first whether it should store the name. If yes, CAP might ask for further information like an attendee's affiliation, department, status (faculty, grad student, undergrad) etc. A THEO entity that describes the first author, a visting scientist at CMU who is in Tom Mitchell's group. After having acquired the values for the slots the entity looks like

```
Bocionek
    Position:  visiting-scientist
    Group?:  no
    InTomsGroup?:  yes
    AtCMU?:  yes
    Generalzations:  visiting-scientist, tom-group-member
    Office:  Weh3508
    Host:  Mitchell
    NumberofYearsAtCMU: 1
```

A typical learned rule, that is not learned by DBL but by induction, may then look like

```
IF ?AttendeeStatus = Undergrad
AND ?AttendeeDepartment = ComputerScience
THEN ?Duration = 30
```

The rule learning of user's preferences is a good example of the combination of DBL (to acquire the necessary basic fact knowledge) with automated induction algorithms.

3.2 Proposal on how to Support Rule Learning in CAP

As a further application of DBL in the context of automated rule learning, we propose to "talk with the user" about the correctness of the learned rules, i.e. how far they meet his intentions. One simple scenario is that, maybe each morning, CAP could ask its user

[6] Advice on date and time does not fully work yet.

[7] ID3 as well as back-propagation are used; see references [11] and [8].

```
Do you want to analyze what I've learned last night?
```

If he wants, CAP would present each learned rule in a readable English-like form and ask whether that is the user's intention. If he says yes, the rule can gain a very high *confidence factor*.

By this mechanism the applicability of a new rule can be determined already after very few examples. This approach, in our opinion, will help CAP especially to learn correct rules for the advice of time and date of a meeting faster, because these parameters don't follow regularities that nicely to detect as those of duration and location. We plan to provide a first mechanism of that kind for CAP as one of the very next steps (see chapter 7).

3.3 Acquisition of Alias Lists in CAP II

Another example of fact acquisition through dialogs is the extension of global alias lists for names and email addresses in CAP II. CAP II, the successor of CAP, is intended to be not only an apprentice but also an agent system with the capability of negotiating meetings *autonomously* by sending and receiving email messages.[8] Parties during the negotiation can be other CAP II agents, but also people without that system. Each CAP II agent parses the semi-structured messages in order to recognize the meaning. There is no natural language understanding component involved. Some sophisticated matching strategies for predefined keyphrases and keyword-value pairs suffice as pointed out in [1].

The extension of name and address list is necessary because, due to various mailers and routing systems as well as personal `.forward` file entries, an email message sent to

```
jdm@cs
```

- known as John McDermott by the agent - may come back as

```
.@cs.cmu.edu:"accent::mcdermot %airg.enet.dec.com"@crl.dec.com.
```

How shall the agent find that correlation? Our solution is a database of alias names and alias addresses that is maintained by the agent. One initial name and email address is always acquired (as one of the properties of a person) when the name is mentioned the first time in an add-event dialog (see above).

When the agent receives a message and cannot determine the name from the address, it *asks the user* by presenting the address and all attendee names involved in the negotiation. Having been told the correct name, the agent adds the unknown address as a new alias to the database. After that, it will never ask that question again.

This application of DBL for the extension of an agent's knowledge is one way to improve its capability of recognizing arbitrary email addresses. The user can hardly provide all imaginable addresses of all possible partners in advance. If he even were able to model the naming strategies of all mailers and routers in the world, it would not suffice because every email user can freely choose nicknames, abbreviations, and forward entries. Another possibility, however, would be the use of external knowledge bases like the UNIX finger command or access to login account

[8] The current status of CAP II is experimental; CAP II has not yet been tried out by all CAP users; only the first author experimented with it. Also the interface of CAP is not yet successfully connected to CAP II due to some technical problems.

lists of the system administrators or other personnel databases (see the ideas for future research in [1] for a discussion of that point).

4 Learning the RAP Automaton with DBL support

The acquisition of facts, as described in the chapter above, is not the only way DBL can be used for learning. What one desires, is, of course, also the learning of *procedural encodings* (see [3], p. 12). The major problem is the transformation of the user's answers to such encodings, i.e. the *operationalization* (see [26], p. 367).

When experimenting with our CAP II calendar agent, which is based on communicating finite automata which interact according to a contract net protocol [1], we found the following knowledge chunks and functions that should be learnable:

- The database that contains information about persons, events and rooms (or how to access it in CMU databases).
- The messages to be generated and at which state to generate them.
- The "vocabulary" for the FSM, i.e. the keywords or keyphrases that decide the next transition.
- The finite state machine (FSM) that decides what to do after receipt of which message (or command).

Our goal is to let the *user incrementally complete* an "empty" apprentice system only by DBL and embedded PbD support. The application that we chose was the room apprentice RAP (see [1], chapter 7.1). In the RAP project, a learning agent for reserving meeting rooms has been implemented. It watches a person when sending email messages to the responsible persons at CMU and extracts (i.e. learns) the "protocol" of that task incrementally. Imagine, for example, that a secretary mails to the responsible "room administrator" in the CS main office to make a reservation of a room for a certain day. She or he gets back the answer that no room is available on that day. Then she or he tries the same with the "room administrator" of the robotics faculty, and so on. The RAP agent will watch those email "dialogs" and learn the knowledge chunks as enumerated above.

We claim that learning the RAP task without support from the user (through dialogs or dialog-like interaction) is almost impossible, or requires very expensive natural language understanding capabilities.

The status of RAP, although fully implemented as described in this report, is still experimental. The most important part missing is a good user interface. What still has to be explored is the acceptance of the "real" user, i.e., whether he is able and willing to answer RAP's questions correctly and consistently. This point will be discussed more deeply in sections 4.5 and 7.

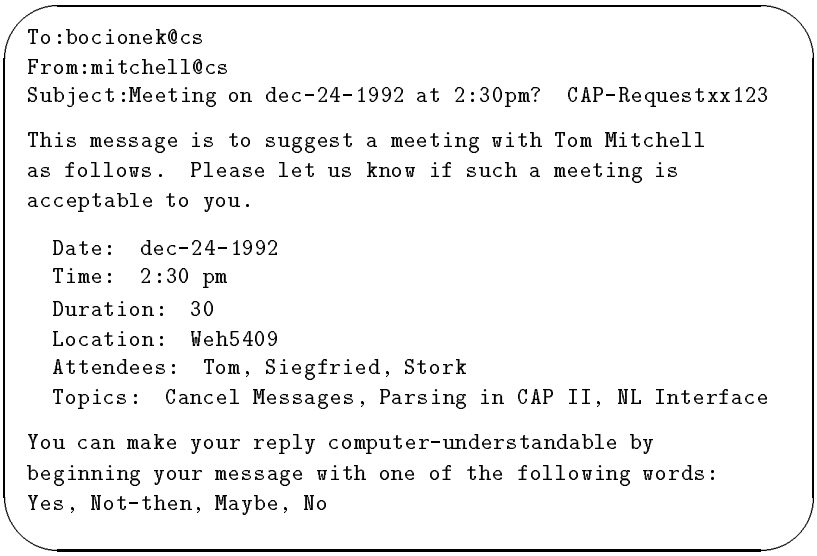

Figure 1: A meeting proposal generated by a CAP II agent

4.1 The Restrictions of RAP

In the RAP project, we follow the client-server model that was successfully used in CAP II.[9] The secretary, who needs a room, is the client. Every room administrator is a server. The following restrictions are defined for their interaction:

- There is only one client that asks for a room for the same event.
- The servers are contacted one after the other (and not all together as the attendees in CAP II).
- The exchanged email shall be as "natural" as in CAP II (see [1], chapter 3.1). An example of a CAP II message – for the proposal of a meeting – is shown in fig. 1. The keyword `CAP-Request` – together with the event identifier `xx123` – in the subject line of the email is used by the agent to filter out the message and relate it to a specific event. The keyphrase in the text part that is used to determine the message type is just the word `suggest`. The data fields like `Date, Time` are used for decisions on how to commit to a meeting or not. At the end of the message a suggestion for people without CAP II agent is given how they manually have to respond.
- Matching algorithms, as in CAP II (see the discussion in [1], chapter 3.4), should be the only methods to analyze email messages.

[9] The host is the client, each attendee is a server. In [19] the according terms are customer and performer.

- Keyword filtering and event recognition works as in CAP II (see [1], chapter 3.1).
- A finite state machine (FSM) should be the underlying operational concept (i.e. no Petri net as, for example, in [14]).

These restrictions have been defined partly because they model the way of room reservation at CMU quite adequately, but partly also to make the task easier for the first try.

4.2 The Initial Domain Knowledge of RAP

```
To:<<address of receiver>>
From:<<address of sender>>
Subject:<<free text>> RAP-Request<<event-id>>

<<free text with meaning-carrying phrase>>

  <<datum-1>>
  << .  .  .>>
  <<datum-n>>

<<Hints or information for non-RAP users>>
```

Figure 2: The meta-template for email messages in RAP

As stated above, the goal of RAP is to allow users with little or no programming skill to complete an "empty" apprentice system. "Empty" means that there is some predefined fact and procedural knowledge in RAP from the beginning, but not enough to perform the task of room reservation in an automated way. The following chunks of knowledge are initially put in RAP:

- **Message format and data fields**
  The meta-format for email messages, derived from the CAP II experience (see above), is depicted in fig. 2. A particular example for a room request is shown in fig. 3 and explained below. The data fields `Date, Time, Duration, Seats, Speaker` are initially known in the current version (but could also be acquired). They are stored as a list of strings in the action templates.
- **Fixed number of classified actions**
  Only a few actions are possible for RAP. In order to facilitate the re-use strategy described below they are classified as follows:

- system: `delete-event, definitely-notify-user, halt-RAP`
- help-user: `send-email`
- disturb-user: `notify-user`

For each action a predefined script-like template exists that describes what is to be done in which way.

- **Augmentation policies for actions and states**
  Actions and states in RAP are augmented with some history information to support the automated activity after learning. For this purpose, disturb-user action scripts contain a slot `from-chains` which is filled by the agent with pairs (state-before, action-before), i.e., which action in which state was performed before this particular action. Similarily, states are augmented by the same pairs, i.e., the history is kept what action and state predecessed the current one. (The augmentation is necessary because FSMs "forget" – once having entered a state – what they performed before.)

- **Learning strategy**
  Learning strategy for RAP means, firstly, the learning mechanism, i.e., dialog-based learning (DBL) only. Secondly, it includes the moment and way when and how a new state is added to the FSM; and thirdly, the moment and way when and how states and actions are constructed and augmented by history information. The whole algorithm will be described later.

- **Re-use strategy**
  To automate the room reservation task (i.e., to re-use the learned workflow knowledge) RAP follows the strategy:

  - Do what user or system want to be done; in other words, perform the system and help-user actions without trying to make any additional or other steps autonomously.
  - Try to avoid what might disturb the user; in other words, try to avoid disturb-user actions and, instead, try to proceed with the task by examining the history information in the augmented states and actions and performing the next required step(s) automatically.

- **Parametrization for re-use**
  To automate a repetitive task, such as sending the same room request to different secretaries, a RAP-like apprentice has not only to re-use the learned FSM, but also to parametrize some actions (with the name or email address of the secretary in the RAP case). The email address is the only such parameter in RAP which is defined as part of the initial knowledge. The possible values for this parameter, however, are acquired during the "whole life" of RAP (see the discussion in 4.5 for the problems with more than one such parameter).

Given the initial knowledge as described in the list above, the user of RAP can now "complete" the apprentice, i.e., build up the thesaurus for the possible message types and construct the

FSM which is the software system *particularily required for his environment*. He does not need to have any particular programming knowledge or skill for that. Just answering the questions of RAP is enough.

4.3 Learning the Vocabulary for RAP

When looking at a typical email message as generated by a CAP II agent one recognizes that only one phrase, sometimes only one word, defines the *type* of the message, like proposal, confirmation or user-cancel-request.[10] Fig. 1 shows an example of a proposal. The meaning-carrying phrase is "`suggest a meeting`". In the context of CAP II, "`suggest`" alone is even sufficient.

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

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

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


You can make your reply computer-understandable by
beginning your message with one of the following words:
No or a room number
```

Figure 3: A request for a lecture room as generated by a RAP agent

According to that phrase (or the message type derived from it), the CAP II agent decides what to do next. This means, mapped to the terminology of finite automata, the phrase is an element of the *vocabulary* of the FSM. This vocabulary can be *learned* incrementally by means of DBL. Fig. 3 shows a message of a RAP agent. The meaning-carrying phrase here is "`need a lecture room`".

From the phrase "`need a lecture room`" the agent can determine the *type* of the message if it "knows" the phrase. Here the type is *room-request*. Each time the RAP agent receives a message where it *cannot* find out the type, it shows the message to the user and starts the following dialog:

[10] The complete set of message types handled by CAP II is described in [1]), chap. 3.2.

`To which type of interaction does this message belong?`

`room-request, positive-answer, negative-answer, alternative-proposal, other`

Assume the user answers with "`room-request`" (under some "good-will assumption"; see [1], chapter 3.1.2). Then RAP will continue asking

`Which is the meaning-carrying word or phrase in that message?`

The user might answer "`need a room`" or only "`need`". If he only says "`room`", the meaning of the email is not defined because there will probably also be messages that contain something like "`have no room`". A solution to that problem might be to continue asking the user until a *verb* is recognized. However, this requires the use of dictionaries and a kind of NL parsing of the phrases given by the user (for example, `need` might be a verb or a noun; the category can only be determined from the context). In our current implementation, we assume that the user answers with a phrase containing a verb. Thus, we can avoid NL parsing (for the moment).

If the user had answered "`other`" to the first question, RAP has to learn also a *new type*, e.g. the type "cancellation" from a phrase "`...lecture is cancelled...`".

From the answers to such questions the RAP agent builds up a database. That database is not only some kind of dictionary; it must be a *thesaurus*. All phrases that the user classifies according to the same type will be treated as *synonyms*. This is *obvious* because human email senders will not always use identical phrases for the same type of message.[11] However, the agent does not need to ask always. If it sees an unknown message, it might first use searching and matching techniques as described in chapter 3.3 for the acquisition of alias lists in CAP II.[12] Also a combination is possible, i.e. the RAP agent first tries to solve the problem itself and then asks the user only for a confirmation.

The hints at the end of fig. 3 are for the room administrator who sees that message the first time. If s/he follows the suggestion, the RAP agent can *autonomously* decide to which type of message the *response* belongs, even if it has *not yet seen* such a message. It perhaps will ask the user for the meaning-carrying phrase too, but *not* for synonyms, because it knows the type from the answer word.

4.4 Learning the Finite State Machine for RAP

The FSM (finite state machine) of RAP can be seen as a two-dimensional table with the *states* as names for the rows, and the *message types* – which might be also *key phrases* – as names for the columns. An entry FSM[state-x, message-type-y] consists of an action list and the name of a state (that can be state-x itself). The action list contains pointers to the augmented action scripts as described above.

When the RAP agent determines a known keyphrase in an email message, it executes the actions in the action list of the corresponding entry. Then, it enters the state that is defined by FSM[state-x, message-type-y]. If an action is of type disturb-user (only `notify-user` is of

[11] For example, in fig. 3, the sender may also have used "`is a room free`" instead of "`need a room`" with the same intention.

[12] In fact, alias lists and a thesaurus are somewhat the same thing.

	EOF	stop-RAP
global-start	–	definitely-notify-user goto global-stop
global-stop	halt-RAP goto –	–
finish-negotiation	delete-event definitely-notify-user goto –	–

Table 1: RAP's empty FSM

that type right now; see above), RAP tries to avoid it and perform the next step(s) of the task automatically.

Table 1 shows the "empty" FSM with the states as index of the rows and the vocabulary as index of the columns. The only phrase that may be received by the "empty" FSM is `stop-RAP` which causes RAP to stop and exit. The column under `EOF` defines a kind of "epsilon transition", i.e., which clean-up actions have to be done after RAP had been stopped (in state global-stop), and after RAP had finished a particular negotiation (in state finish-negotiation).

After having seen the RAP request message (as showed in fig. 3) and an example of a positive answer, the FSM will be as depicted in table 2. It now consists of four states (`global-start, global-stop, finish-negotiation, s1`) and four "understood" message types (*EOF, stop-RAP, room-request, positive-answer*). Note the empty fields FSM[global-start, positive-answer] and FSM[s1, room-request]. Their contents will be synthesized when those cases occur, e.g. a positive answer for a request is received "before" sending the request. This can only be the result of some error, like duplicate email sent by the room administrator etc.

If, after having constructed the column for *room-request*, RAP sees an email message of this type, it performs the action "send email" which forwards the email message to the specified room administrator. After that, RAP enters state `s1` as indicated in FSM[global-start, room-request].

	EOF	stop-RAP	room-request	positive-answer
global-start	–	goto global-stop	send-email-request goto s1	
global-stop	halt-RAP goto –	–	–	–
finish-negotiation	delete-event def-notify-user goto –	–	–	–
s1	–	goto global-stop		def-notify-user-pos goto finish-negotiation

Table 2: A little part of RAP's FSM

The *learning task* for RAP, therefore, is to construct a FSM like that in table 2 by watching all outgoing and incoming email with a RAP-Request keyword in the subject line. Initially, the FSM knows the three states*global-start, global-stop, finish-negotiation*. When RAP is started, it enters *global-start*. The only known keyphrase is *stop-RAP* that can only come from RAP's owner. In each state-x (also in later acquired ones), jumping to state *global-stop* will be the only action in the action-list of FSM[state-x, "stop-RAP"].

New states are constructed (see also section 4.2) when

- no transition can be found in the automaton,
- no transition can be derived from the history information in the augmented states and actions, and
- the user answers with `no` to the following question:

  ```
  Is this the last action of the event (yes/no) ?
  ```

  This is the case when RAP receives a `room-request` message the first time (being in state `global-start`). Then, state `s1` is introduced in the FSM.

The contents of the field FSM[s1, positive-answer] in table 2, i.e., the action `def-notifyuser-pos`[13] and the transition `goto finish-negotiation` was filled in after having answered the question above with `yes`. The action `def-notify-user-pos` was derived from the `definitely-notify-user` action template. It must be "definitely" because the user wants to know the positive answer from the room administrator, i.e., the selected room number. Otherwise, RAP would try to avoid the notification as a kind of user disturbance (see section 4.2). The transition, `goto finish-negotiation` is also predefined after the user answered `yes` to the "last action" question.

[13] That's an abbreviation for `definitely-notify-user-positive-answer`.

0:	Init: all following variables are set to nil here	
1:	Take RAP email ?m	[see example of fig. 3]
2:	Determine event-id ?id from ?m	[here: "xx123"]
	IF ?id = nil	
	THEN ?m.id ← ?id ← generate-id ()	
3:	Determine current state ?s-old from ?id	[currently: "global-start"]
4:	Determine message type ?t from ?m	[not possible since unknown type]
	IF ?t ≠ nil	
	THEN	
5:	?s-new ← FSM[?s-old, ?t].goto	
	IF ?s-new = nil	[i.e., FSM field is empty]
	THEN	
6:	?act ← find-action-triggered-by-message (?t)	
	IF (?act)	
7:	THEN ?s-new ← take-trans-from-act (?act)	
	ELSE	
8:	?s-new ← add-new-state-to-FSM ()	
	FSM[?s-old, ?t].goto ← ?s-new	
9:	?act ← derive-action(?last,?m,?s-new,?s-old)	
10:	add-to-action-list (?act, ?s-old, ?t)	
11:	perform-actions-and-goto (?s-old ?t ?m)	
12:	try-to-perform-epsilon-transitions (?s-new)	
	ELSE ; use DBL to find out ?t	[DBL becomes necessary]
13:	?t ← ask-user-for-message-type (?m)	["To which type ... belong?"]
		[The user answers with "nil"]
14:	?p ← ask-user-for-keyphrase (?m)	["Which is ... phrase?"]
		[The user answers "need a room"]
	IF ?t ≠ nil	[That is not the case here]
	THEN	
15:	add-synonym-to-thesaurus (?t ?p)	
16:	<< perform steps **5. - 12.** >>	
	ELSE ; make a new message type ?t from ?p	
17:	?t ← ?p	[Now new type "need a room"]
	add-type-to-thesaurus (?t)	
	add-keyphrase-column-to-FSM (?t)	[Column "need a room" added]
18:	?last ← ask-user-for-last-action (?m)	[was this the last action?]
	IF ?last	[That is not the case here]
19:	THEN ?s-new ← "finish-negotiation"	
20:	ELSE ?s-new ← add-new-state-row-to-FSM ()	[Row s1 is added to the FSM]
21:	FSM[?s-old, ?t].goto ← ?s-new	
22:	<< perform steps **7. - 12.** >>	

Table 3: The algorithm of RAP's learning FSM (example in brackets)

The algorithm for the learning of RAP's FSM works as sketched in table 3. The comments – in brackets on the right side of the algorithm – explain what happens with the empty FSM of table 1 after receiving the email message from fig. PIC-request.

1. A RAP email message `?m` is read.
2. The agent tries to find out the event-id `?id`. If there is no event-id in the message's subject line, it creates one and adds it.
3. The current state `?s-old` of the event with event-id `?id` is determined.
4. The message parsing and learning part starts with the search for the keyphrase, i.e., the message type `?t`. This is done by searching in the message and matching the words and phrases of the thesaurus.
5. If `?t` exists, the agent looks up the transition to the new state `?s-new`.
6. If `?s-new` does not yet exist, RAP tries to find an action (from the history information) which was already triggered by the message type `?t` somewhen before.
7. Supposed such an action `?act` exists, RAP can just use the transition information which is stored as a part of the augmentation of `?act`.
8. If `?act` could not be found in the history, the new state `?s-new` has to be created and added to the FSM. It is the target state for the current transition.
9. Furthermore, the necessary action `?act` has to be derived from the action templates as predefined in the domain knowledge. If the email `?m` came from the user, the action wil be a derivative of `send-email`; if it came from outside, a derivative of `notify-user`.
10. The now determined action `?act` can be added to the action list of the current FSM entry.
11. In the function `perform-actions-and-goto` the action list of the current FSM entry is performed according to the strategy described in section 4.2. "Help-user" actions are executed, "disturb-user" actions might be skipped and follow-up actions automatically started. After finishing all actions, the transition to the new state as indicated in `FSM[?s-old, ?t].goto` is performed.
    Note that the augmentation of state and action descriptors with history information takes place always at the end of each performed action.
12. Having done the state transition, RAP tries to perform immediate (= epsilon) transitions if there are some (can be found in the EOF column of the FSM table, see above).

The following steps (13. – 22.) take place if RAP was not able to find out the "meaning" of the email message `?m`. Then, the dialog-based learning (DBL) mechanism starts.

13. First, the user is asked for the message type `?t` as described in the last section.

14. Then, he has to indicate the keyphrase `?p`.

15. If the user was able to name a known message type for `?m` (i.e., `?t` $\neq$ `nil`), the keyphrase `?p` is just a synonym for the message type `?t` and can be added to the thesaurus.

16. Then, RAP can process the email message exactly as described in the steps 5. – 12.

17. In the case of `?t = nil`, the phrase `?p` is made a new type `?t` and added to the thesaurus. In addition, a new column is created for the FSM that will hold the transitions and actions for that new message type (a `need a room` column in the example).

18. Before the new state can be determined, the user must be asked whether the email message `?m` was the last action of the event.

19. If so, the new state `?s-new` will be `finish-negotiation` (where some book-keeping of RAP will take place as well as a final notification of the user).

20. If not, a new row has to be added to the FSM with a system-generated name for `?s-new` (`s1` in the example).

21. Then the state transition `FSM[?s-old, ?t]` $\leftarrow$ `?s-new` can be defined.

22. At that point, learning is finished and the execution of the action list as well as the state transition can be performed as described in the steps 7. – 12.

Note, that the "ask-user-for-..." function calls always have the complete event descriptor (including the FSM) as an implicit parameter. Note also, that these calls may start with some autonomous match and search activities in order to avoid a disturbance of the user (see also section 4.3 above).

4.5 Discussion

Our intention in developing the RAP system is to explore the power and applicability of DBL in office automation systems that are mainly used by people with no or little programming skill. The objective is to give them software that becomes more useful during its usage. Its easy-to-handle interface for textual dialogs, that we described above, allows also this user group to "program" new functionality [2], i.e. the capability of the software can permanently be *extended*.

- First, it helps the system to increase its fact knowledge and perform tasks with less disturbance of the user.
- Second, the "tolerance" to different variants of email messages that have the same semantics can be enlarged.
- Third, the concept of PbD allows us to stepwisely extend also the workflow model built in office automation software by adding new states and transitions to the automaton that represents the interaction of people during an office task.

However, without the support of some kind of DBL interface, a workflow model – like that of RAP – could probably *never* be acquired by only analyzing in and outgoing email.

The advantages of office software that contains DBL mechanisms for permanent improvement are obvious. One can develop field-customizable tools that adapt themselves to the specific needs of a particular end user. This is an important issue if one wants to reach a wide-spread acceptance of the software. From the developer's point of view such software requires less programming time because it needs not contain all "imaginable" functions any user might ever require.[14] The basic software can be sold to more customers because their different requirements can be met during the tool usage. Nevertheless, a lot of questions still remain open.

- **The natural language question:** Is matching and searching of keyphrases (as discussed in [1]) really sufficient to detect new workflow concepts? Or must we go the hard way to natural language understanding? Or how can we mix both techniques successfully? As mentioned above, a distinction between verbs and nouns may be required, but is quite impossible without context analysis. Another question is how to detect the relevant data variables from free text if there are no nicely structured key-value pairs with predefined keywords.

- **The task complexity question:** RAP as an appliciation of DBL for the acquisition of workflow concepts is a rather simple example. Can we also apply the approach to detect the workflow of much more complex office tasks like the meeting negotiation as "hard-coded" in the CAP II agent system [1]? How will the extension of the FSM work if more than one response is expected, i.e., the answers of more than one attendee in CAP II. Then, a kind of "answer collection" must be performed until the user proceeds with his next activity. To make it even harder, the responses might be different, like yes, no, or maybe in CAP II. Will a RAP-like learning mechanism be able to learn the required branching? (In fact, the "reprogramming" of the CAP II system by means of a RAP-like interface is one of our planned activities; see section 7.)

- **The automated parametrization question:** In RAP, only the parameter `to-addr` for the email address of room-administrating secretaries is used when the search for a room is performed autonomously. Therefore it is sufficient to try out all known/learned addresses sequentially. For meeting negotiation in CAP II, however, at least the parameters `date` and `time` must be considered. What is a good strategy to select values for them? Simple Prolog-like enumeration, specific heuristics, or what else?

- **The FSM representation question:** How shall we deal with office procedures where an FSM representation is not sufficient? For example, the processing of customer orders might require negotiation with the customer as well as with some part suppliers. These kinds of parallel, synchronized workflows will probably require Petri-net like representations. Can we *generalize* our DBL-driven acquisition techniques for automata structures in order to learn such nets too? How shall this be done? How must the history be stored and organized in such cases? What if the amount of history information "explodes"?

[14] Also if so, there will probably be someone who has some further need.

- **The authority transfer question:** Without field tests it can also not be decided whether the user will accept that an agent constructs and sends email messages on his behalf. In the case of room reservation it might be easy to convince him. But in the case of making commitments to or declining participation in meetings this question is really important. Shall the user be asked every time before sending a message or can he transfer the authority, at least for a specific period of time (like his vacation)? Can the agent perhaps learn when to ask and when not? Answers to those questions might well be key issues for the final acceptance of such a system.

- **The "consistent user" question:** A critical point is whether the user always acts consistently, i.e., performs the same actions in the same *syntactical* situation. How shall the apprentice deal with inconsistencies? Can a reliable behavior learned from such a user at all?

Our RAP implementation is no ready-to-use tool (yet). It should be seen as a *first experiment* towards the development of office automation software that adapts to its users during operation. Since it deals with a real task in the daily office work, we think it is a good candidate to study the questions mentioned above step by step. In addition, the availability of – and the experience with – our hand-coded meeting negotiation apprentice CAP II (also still experimental) gives us the opportunity to apply our RAP ideas to an even more complex task and to evaluate new solutions in RAP, under some generality aspects, in the context of the CAP II task.

5 DBL as a Support for Programming by Demonstration

In this chapter we want to describe how DBL and PbD can be combined in order to acquire procedural knowledge in the graphics editor domain. This approach is different from the previously shown acquisition of knowledge or the construction of an automaton. In contrast to these applications, the PbD system does not apply the newly learned procedures automatically. They are kept and can be selected from a menu, such that the user may apply them when needed.

The acquisition of new procedures is based on PbD that enables even a computer novice – whom we can assume is not able to understand or even program tasks in a regular programming language like C or LISP – to demonstrate procedures for a graphics editor by the demonstration of traces of basic actions without explicitly writing code. Providing a PbD system[15] with multiple examples in the form of action sequences as the input, it should be able to generate a procedure that implements the demonstrated task. Nevertheless, this approach is based on DBL as the key solution in order to overcome the problem of PbD to meet the user's intentions.

A second application for DBL within PbD – besides the learning of new procedures – is the possibility to cope with the incompleteness of the domain theory as addressed in [31]. The same techniques which we propose in order to learn procedures, can also be used to support the extensibility of the domain theory. This is done by the learning of new property and relation functions[16] "by demand", i.e. if the existing ones are not sufficient to "find the meaning" of a trace or to generalize the intended procedure. As we will show in this chapter, the combination of PbD and DBL is a general approach in order to generate procedures within interactive systems.

5.1 The PbD Concept for the Graphics Editor

Fig. 4 shows the principal structure of a PbD system that is able to acquire procedural knowledge from traces of actions. This approach is not limitated to the domain of a graphics editor but can support all interactive systems.

The demonstrated *event* is provided in a fixed syntactical form, which is called *event description*. The event description consists of the basic actions and user-defined procedures of the graphics editor: For example, the selection of graphical objects, the changing of line or text modes, and the deletion of objects.

A set of expert-supplied *property and relation functions* is used to find out the "meaning" of an event. This step is called the *event interpretation*. It is equivalent to the extraction of its semantics. In our case, the property and relation functions extract geometric and spatial information such as angles, lengths, orientations, crossings or the relationship of objects in the working window. The result of the interpretation is called the *event representation*. It is composed of the conjunct of all valid, instantiated property and relation functions applied to the event description (which may be represented as a feature vector) and the event description itself. Consequently, the event representation contains the complete semantic and syntactic information of an event with respect to the set of property functions.

[15] A PbD System is a program that realizes Programming by Demonstration.

[16] Property and relation functions are used to interpret the demonstrated traces. They build up the model of the domain.

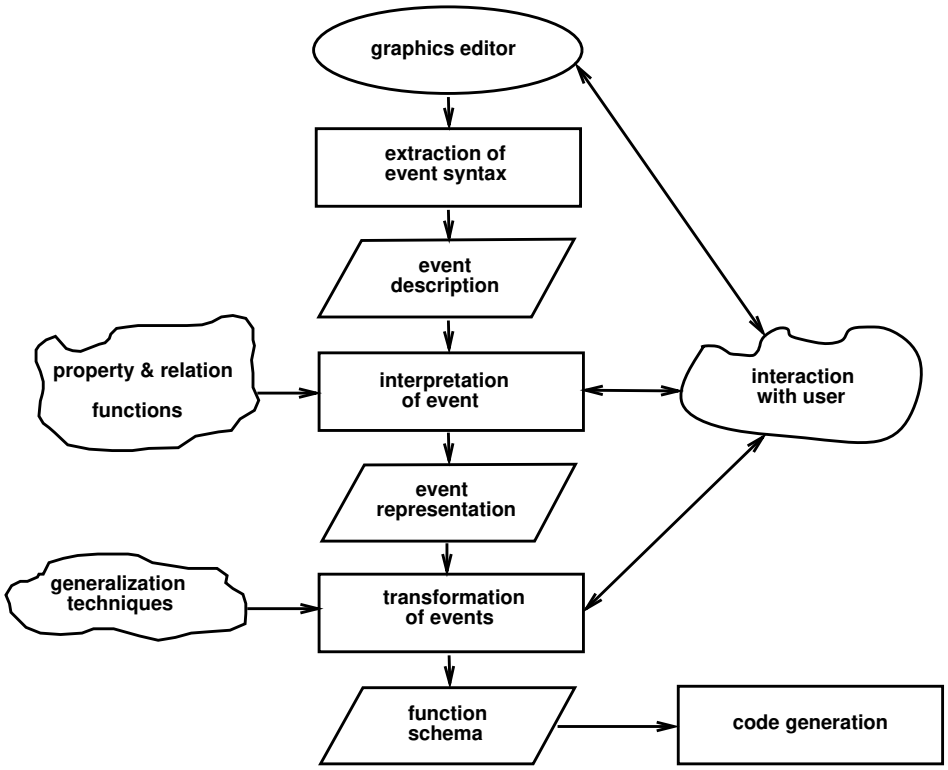

Figure 4: PbD concept for a graphics editor

The *generalization functions* are used to extract "characteristic features" from a collection of event representations. They synthesize a procedural schema that represents the control and action flow of the demonstrated events. The control part of the schema is derived from the properties of the event, whereas the action part is extracted from the action trace.

Since there is no fundamental difference between property and relation functions and procedures, new property and relation functions can be defined based on PbD. Included in the PbD system they help to find the meaning of an event easier and faster.

5.2 Problems

Although the concept of PbD looks straight-forward, a lot of problems must be overcome in order to enable the user of the graphics editor to enhance it by using this technique.

The major problem is that we cannot provide a complete (or even "enough strong") domain theory that describes and explains all possible actions in the graphics editor in advance. However, this would be desired if explanation-based learning (EBL as defined in [24] and [7]) had to be applied. EBL looks promising because it can find an explanation in terms of predefined schemata of the domain theory from one or only a few examples. In our case it is not only impossible but also undesirable because a user of a graphics editor might wish to construct a potentially unlimited number of procedures at some point.

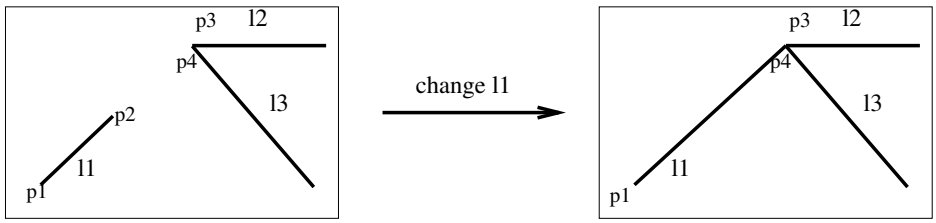

Figure 5: Ambiguity of explanations

Since we cannot, and do not, want to define a strong domain theory, we have to work with an incomplete one as provided by the initial set of property functions. The consequence of that lack is that the PbD system cannot (always) correctly decide *why* a particular action was performed. However, in order to find a procedure that describes a demonstrated event, the PbD system needs an *explanation* of all actions of the event in order to find the causality between them. Only if this information is available, can the PbD system synthesize the intended procedure.

K. VanLehn [38] came to a similar conclusion. He pointed out that inductive learning can only be performed when explanations in form of intermediate results are given. He argued that it is very hard for a system to learn more than one thing per example because, otherwise, it cannot analyze the reason why something happened. We want to pick up his idea, extend it and make it applicable for the graphics editor domain.

Let us discuss on an example why the lack of explanations is a big problem for induction systems. As shown in fig. 5, sometimes even a very simple action can have ambiguous explanations. Here, we have *at least* six possible explanations for the changing of line l_1:

- connect point p_2 of line l_1 to the intersection point of line l_2 and l_3,
- connect point p_2 of line l_1 to point p_3,
- connect point p_2 of line l_1 to point p_4,
- double the length of line l_1 and "add" the second part straight at p_2,
- change point p_2 of line l_1 by $(\Delta x, \Delta y)$,
- change point p_2 of line l_1 in order to meet point $(300, 330)$.

In this case, all knowledge about a graphics editor cannot help to decide which explanation was intended by the user. The only support the system can provide is a "most probable" (via some predefined metrics or by learned decision rules according to the mechanism used in CAP) explanation for this system or for a special user.

Here is the motivation for our approach to use dialog-based learning, i.e. involve the user in the process of finding an explanation by asking questions. The next section will outline this approach in detail.

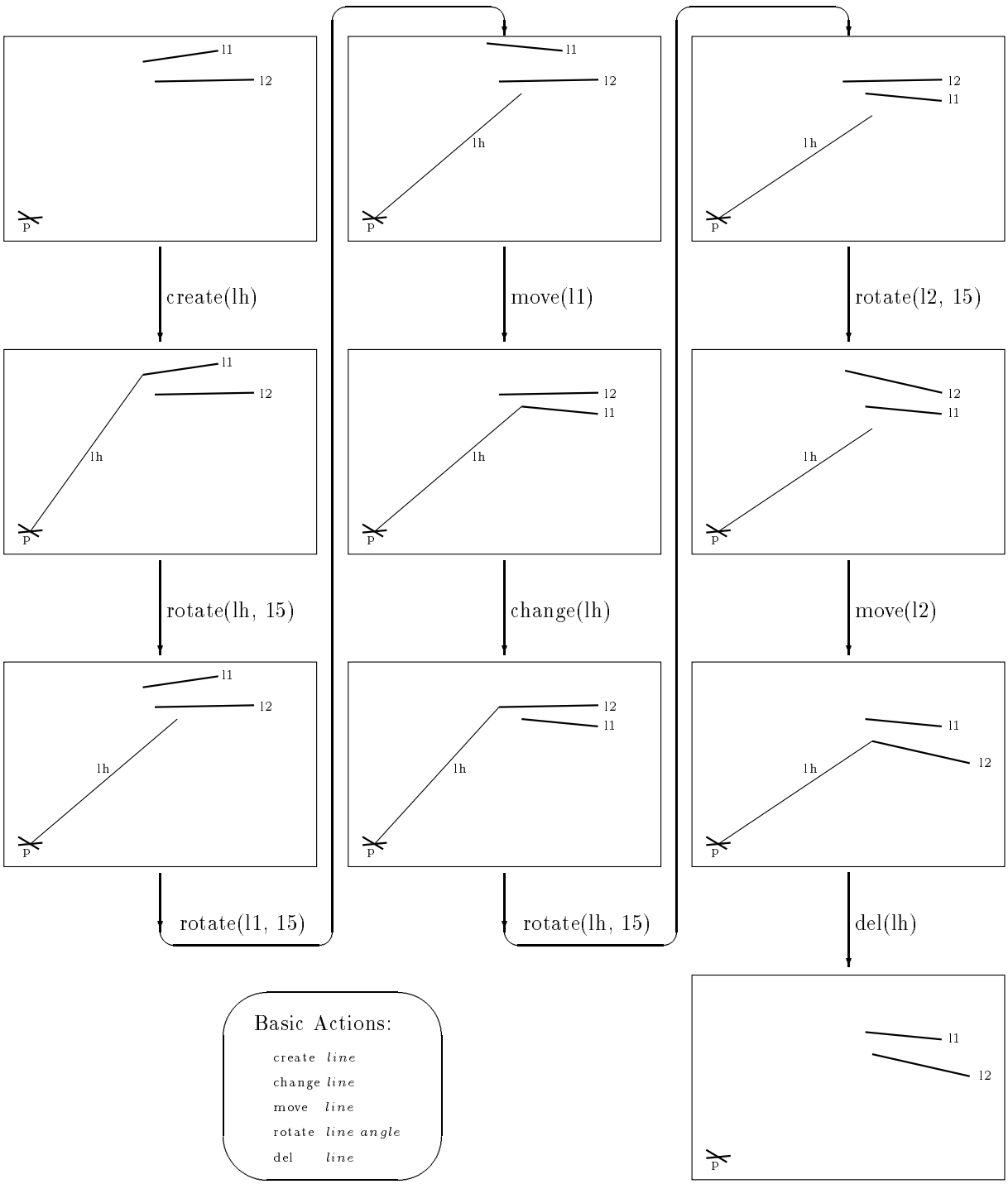

Figure 6: Rotation of objects around a fixed point p

5.3 DBL as a Solution to Handle Incomplete Domain Theories

We have seen that built-in knowledge is not sufficient to decide what a user intended by his or her action. Therefore, we have to enrich the incomplete domain theory with the help of the *user* by providing some DBL mechanisms. The user *is* able to provide the necessary domain knowledge to the PbD system. Especially, he or she knows the intention of the action.

We want to distinguish between two different kinds of dialogs: The *implicit* and the *explicit* dialog.

The *implicit dialog* is used as part of the demonstration of an event. The user of the graphics editor introduces new objects in the graphics editor that *support* the interpretation of the construction. This may, for example, be a line that establishes the relation between two points. In fig. 6 [17] we have used such a line to make a connection between the endpoints of lines l_1 and l_2 with the meeting point p of two other lines. This approach was also taken by D. Maulsby [17] who uses "supporting" lines in METAMOUSE in order to identify the next object that has to be manipulated during a loop.

The introduction of new objects can be used in all interactive systems, e.g. graphics editors, text editors or user interface builders. Real-world applications like robotics do not directly support this form of dialog. However, real-world applications are often equipped with an additional simulation tool that, in general, is based on a graphics interface, so that implicit dialog can indirectly be introduced based on the simulation tool. This shows that implicit dialogs are a widely applicable feature of DBL.

The *explicit dialog* is necessary for expressing intentions that cannot be recalled in the graphics editor domain. For example, in a graphics editor it cannot be expressed that one parameter of an action a_2 is the same as another parameter of an action a_1 performed in the beginning of the demonstration of an event. However, this information can be provided by the user through an explicit dialog.

Besides this differentiation, another classification of interactions between the PbD system and the user of a graphics editor can be defined: *System-initiated* and *user-initiated* dialogs.

If the PbD system cannot find out what the intention of an action was, or it has too many possible hypotheses for a generalization, the system itself initiates a dialog. It can directly ask the user by interrupting his or her work. Another alternative is that the system takes the most probable hypothesis and displays it. Only if the user continues working, this hypothesis is assumed to be correct.[18] Otherwise, the user has to interact with the PbD system. In contrast to D. Kurlander's graphics editor CHIMERA [15], that provides the possibility of re-editing

[17] The figure shows the construction of a rotation procedure by the demonstration of a trace of applications of five basic actions. After processing this event by the help of a PbD system a procedure is formed that rotates every object consisting of lines around a predefined point.

First the help line l_h is created. l_h connects the point p with the endpoint of l_1. Then l_h is rotated around its beginning point by 15 degrees. Afterwards, l_1 is rotated also by 15 degrees and moved so that its end point meets l_h again. The next step is to change l_h so that it meets the end point of line l_2. Afterwards, l_h and l_2 are rotated also by 15 degrees. Next, l_2 is moved so that it meets again l_h. Finally l_h is deleted.

[18] Similar Mechanisms can be found in A. Cypher's EAGER [5] and D. Maulsby's METAMOUSE [17]. Both systems anticipate the user's next action and judge it as ok unless the user cancels the displayed hypothesis.

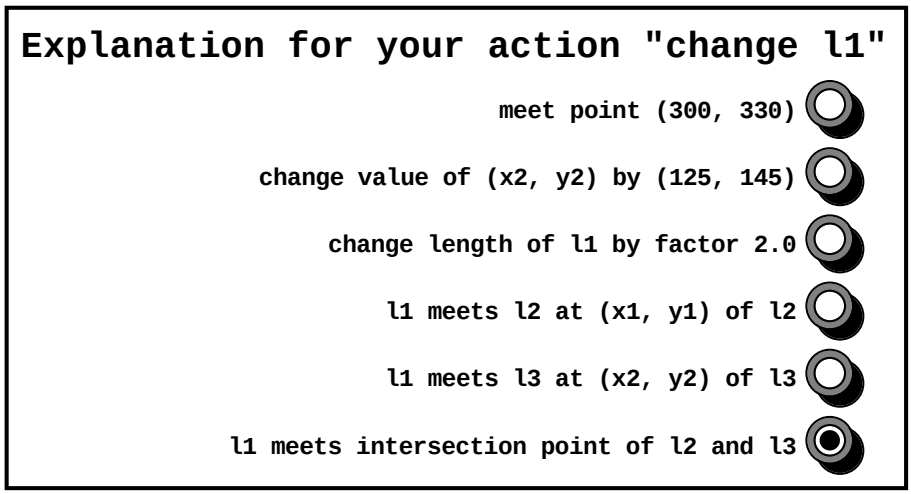

Figure 7: Panel with six explanations of a change-line action

the interpretation of the events *after* the demonstration, the system-initiated dialog can take place during or after the demonstration of an event, depending on the situation or the user's preferences. Fig. 7 shows a list of possible explanations of the "change" action as applied to line l_1 in fig. 5. The user has checked the sixth explanation, i.e. that l_1 shall meet the meeting-point of lines l_2 and l_3.

A *user-initiated dialog* takes place if the user wants to express something that is not inducible from a few examples. For instance, if the user wants to give a branching condition of a demonstrated procedure, it is often better to give this condition in terms of the predefined property functions instead of letting the system come up with all possible hypotheses.

5.4 Embedding of DBL into a Graphics Editor

Next we want to discuss which problems can be solved by DBL and how it should be embedded into a PbD system.

In principle we have three classes of problems. The first class deals with the interpretation of a single action. The second class copes with the problem of how an action is related to other actions or objects of the demonstration's trace. The third class is concerned with how the set of built-in property and relation functions can be enlarged. We want to address all three problems and propose possible solutions for them.

5.4.1 Interpretation of a Single Action

One of the problems of interpreting a single action has been shown in fig. 5. At least five possible approaches can be provided in order to reduce the amount of possible explanations of the "change" action. This are the use of panels, highlighting, additional actions, property and relation forms, and a metacalculator.

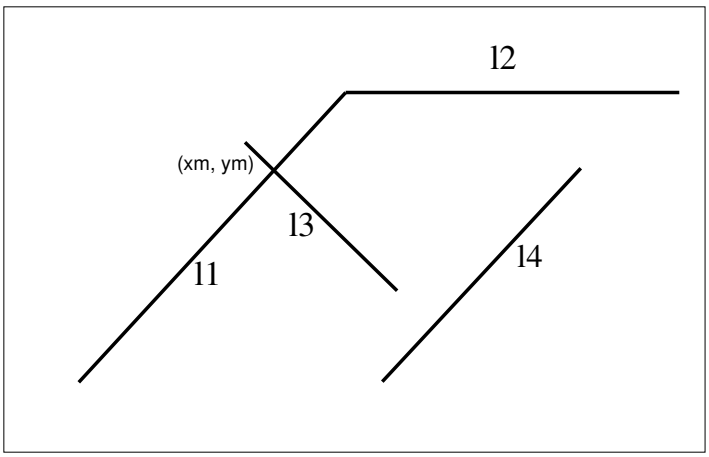

Figure 8: A graphics situation

The first solution is based on panels (as shown in fig. 7), in which the PbD system can *initiate an explicit dialog* by showing the user a list (i.e. the panel) of all possible interpretations expressed in terms of the predefined property and relation functions. If the user wants to select another interpretation, the demonstration in the graphics editor can be stopped and a selection can be made by the user. Otherwise, the interpretation is accepted and the user can continue with any additional interaction.

Multiple selections in the panel should be possible in order to indicate that more complicated (DNF) expressions have to be defined. In order to reduce the hypothesis space for the DNF expressions, additional interaction with the user can help. With questions like

```
How many connections between explanations do you want to be made at most?
```

and so on, the user can further reduce the amount of DNF formulas that have to be presented in case that too many hypotheses are possible. However, it is questionable what kind of knowledge a user must possess to answer this. Perhaps that kind of technical questions should only be asked in a kind of "expert mode" or so.

In order to make this approach applicable for a huge number of possible explanations, we have to introduce the mechanism of abstraction. Instead of presenting every interpretation of the action we only present a list of property and relation functions like *meets-with*, *intersects-on*, *parallel*, *length* or their descriptions *without any* parameters. The user can select all property and relation functions that are important in order to explain the last action. Afterwards, non-selected properties are not taken in account any longer. The level of abstraction can be reduced with the reduction of possible explanations of an action.

The concept of a panel that shows the interpretation is general enough for all applications that are based on the PbD mechanism because it is based on predefined property and relation functions and simple questions that are related to them.

Another very simple but effective means of reducing the amount of possible explanations is the *highlighting* of interesting objects. Highlighting line l_1 in fig. 5 would tell the system that

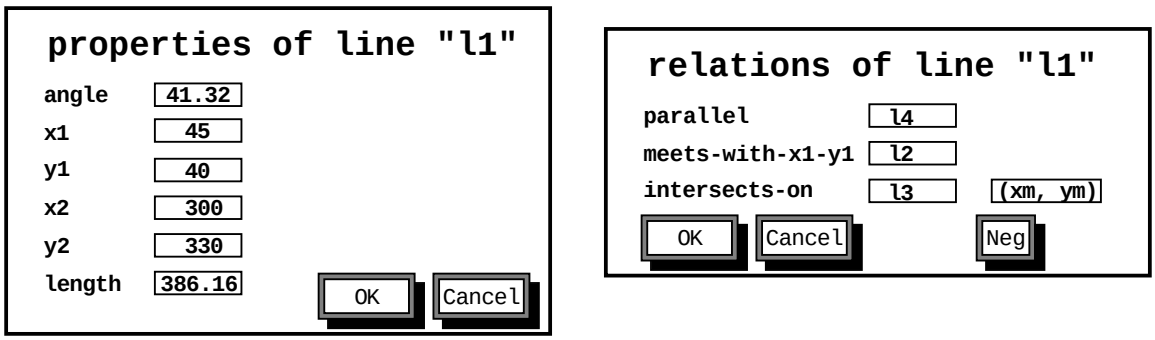

Figure 9: Properties and relations of line l_1

we are interested in the object l_1 only. Then, just the first three of six possible explanations – as shown in fig. 7 – are left over. In case we want to express that the intention of the action is that l_1 meets the meeting point of l_2 and l_3, the highlighting of all three lines is necessary. Highlighting of selected objects can be treated like an explicit dialog initiated by the user. It might also be possible if the user could express that he is interested in *at least* lines l_2 and l_3. This might help to avoid that he forgets to tell the system all objects that are part of a specific, intended goal situation.

The combination of selecting property and relation functions within a panel and objects by using the highlighting mechanism enables the user to drastically reduce the amount of explanations with simple interaction. It is an approach to provide scalability as mentioned in chapter 1.2.

The next way to find explanations for actions is by using additional actions. The principle idea behind this is that the user can give more positive or negative examples of the last demonstrated action. This can be done by a single action or by an action sequence. These local explanations help to drastically reduce the amount of complete examples. By doing this, we provide the system not only with an action trace but even with an action tree.

The fourth way of providing information about the user's intention is by performing a user-initiated dialog via forms. The user can select objects about which he or she wants to have information in terms of the property functions. This information is given using *property* and *relation forms* as shown in fig. 9. There, the forms provide information about line l_1 in the scene of fig. 8.

In order to manipulate the object l_1, the user "tells" the PbD system how to perform that by directly writing new information in the form "properties of line ...". Having done this, the user only needs to hit the return key, and the PbD system changes the values of the object l_1 in the graphics editor. The relations of the object l_1 (in fig. 8 and displayed in form fig. 9) can only be read but not changed, because relations have no inverse function. For example, there exists an unlimited amount of inverse functions that make two non-parallel lines parallel. With the purpose of providing the ability to change relations of objects, the system could show different default inverse functions that can be selected by the user.

Of course, this kind of interaction is helpful for precise manipulations, but not for providing

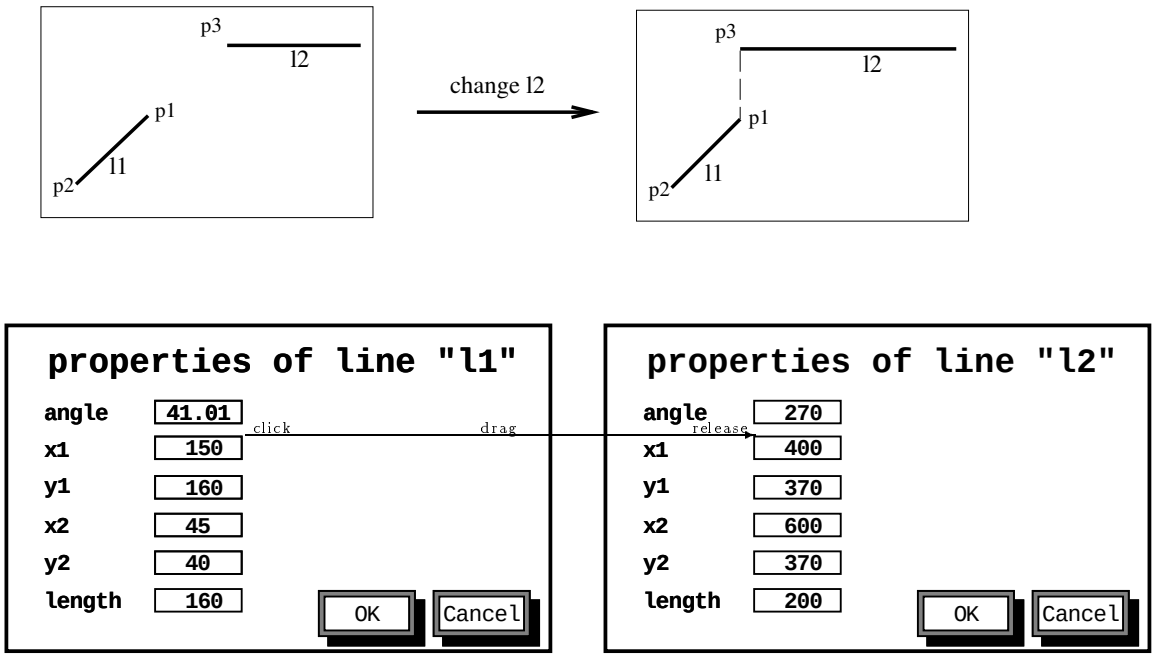

Figure 10: Manipulation with the help of property and relation forms

new explanations for an action. However, it can be applied by introducing a new copy mechanism with a fixed but very useful semantics. If a value of a field of a property or relation form is selected, the mouse dragged to another field of (possibly another) property or relation form and afterwards released there, the value is copied in this field. Additionally, the PbD system interprets the copy&drag as an assignment of variables, not only of values.[19] With this means the user is able to explicitly tell the PbD system that the change of the object was not only done in order to change a specific value but to change this value in relation to another object in the same scene. Although this mechanism is much more *explicit programming* than demonstration, it is a simple form of performing dialogs with the system which transfers the user's knowledge (his or her intention) to the PbD system.

With an additional metacalculator like that in fig. 11 the user can extend the expressional power of this approach. The user has only to collect the necessary numbers from the relevant property and relation forms, transfer them to the metacalculator by using the copy&drag mechanism, and connect the information appropriately.

Examples

Suppose that the user wants to change line l_2 of fig. 10 such that the x-coordinate of point p_3 of line l_2 is the same as the x-coordinate of point p_1 of line l_1. This can easily be done by displaying

[19] This is comparable with the semantics of computer languages. There, it is a difference between $x := a$ and $y := a; x := y$.

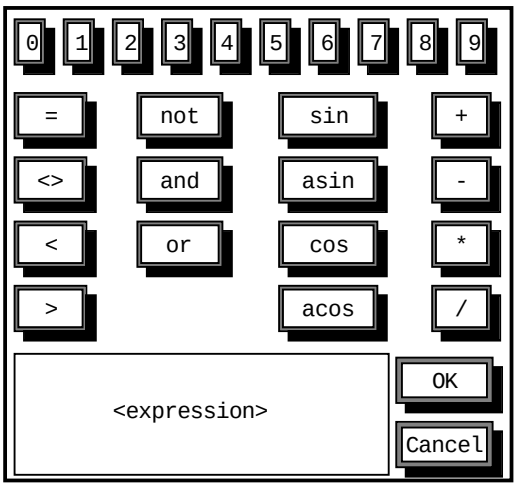

Figure 11: Example of a metacalculator in a PbD system

the property forms of each line and then dragging the x_1-value of line l_1 in the x_1-field of line l_2. With this simple action and knowing the underlying semantics the PbD system is now able to understand the intended operation without any problems.

With an additional metacalculator we can for example, show the system that we like to move the line l_1 nearby line l_2 such that the beginning of line l_1 is exactly 10pts in x-direction and 15pts in y-direction away from the end point of line l_2, we proceed as follows:

First, we introduce a third line l_h in the scene. Then we pop up the property forms of the lines l_2 and l_h. In the next step, we drag the x_2-value of line l_1 to the metacalculator and add 10 to the value. After this we drag the result $(+\ \ x_2(l_4)\ 10)$ to the x_2-field of line l_h. We repeat the last three steps for the y_2-values and the value 15 instead of 10. Because the graphics editor is able to snap in the endpoints of the lines, we are now able to move the line l_1 so that its beginning point meets the endpoint of line l_h. We finish by deleting the help line l_h and removing the property forms.

Another example that can be "programmed" with the help of the property and relation forms is the moving of a line to the intersection point of two other lines as shown in fig. 12.

First, we introduce a help line l_h. After that, we pop up the property form of line l_h and the relation form of line l_2. The next step is to drag the x_m-value of *meets-on* $(l_3\ (x_m, y_m))$ to the x_1-field of line l_h. Then we drag the y_m-value of *meets-on* $(l_3\ (x_m, y_m))$ to the y_1-field of line l_h. At the end, we move the line l_1 to the beginning point of l_h and delete l_h and the forms.

Discussion

Highlighting, panels, property and relation forms, and the metacalculator are approaches to "tell" the PbD system the user's intention explicitly. Although these mechanisms are quite different from the dialogs we have discussed for apprentice systems (see chapters 3 and 4), we call them DBL. The user supports the synthesis of new functions by interacting with the PbD system: Instead through a textual representation of the dialogs, however, it is mostly more

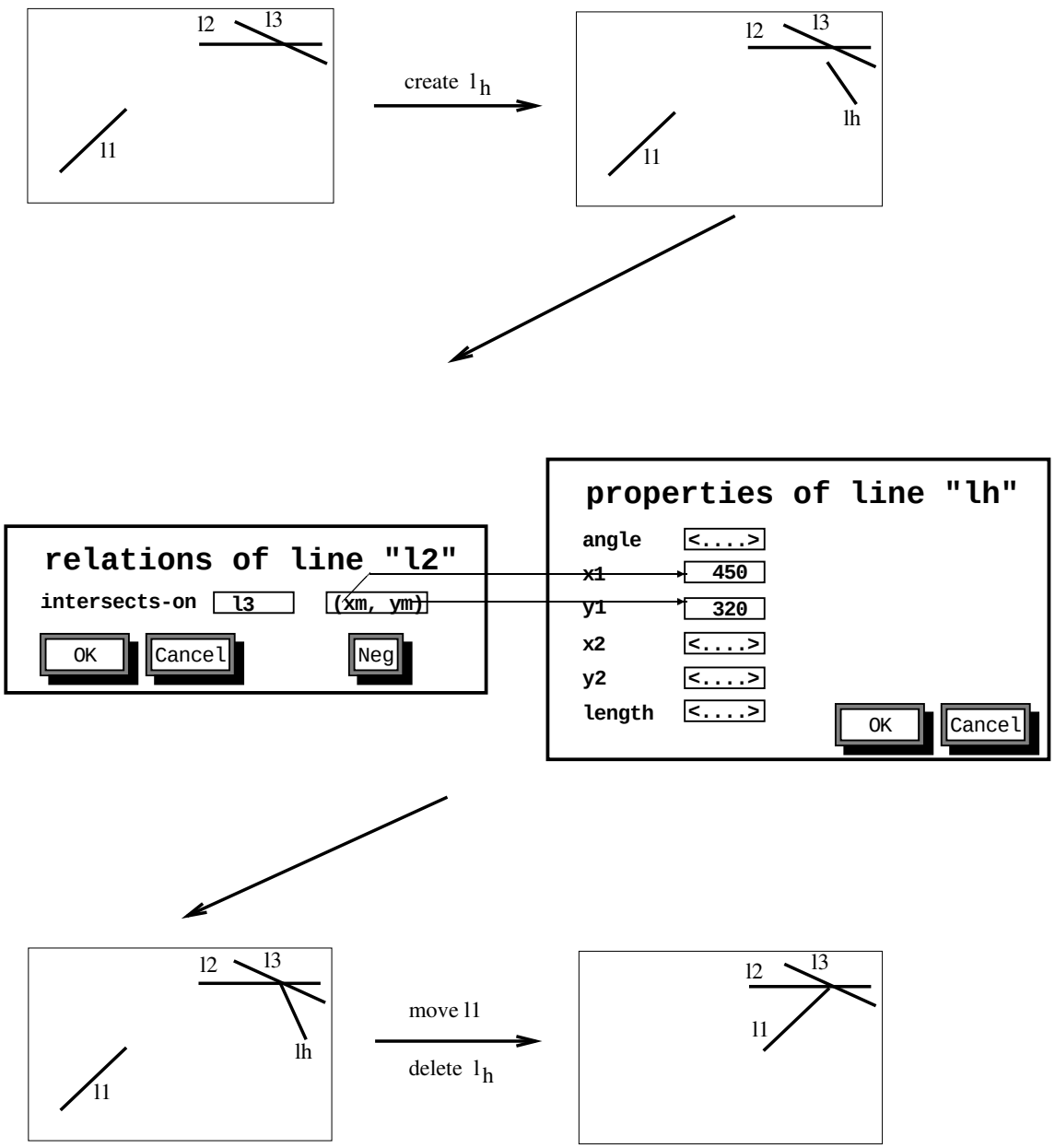

Figure 12: Move a line to the meeting point of two other lines

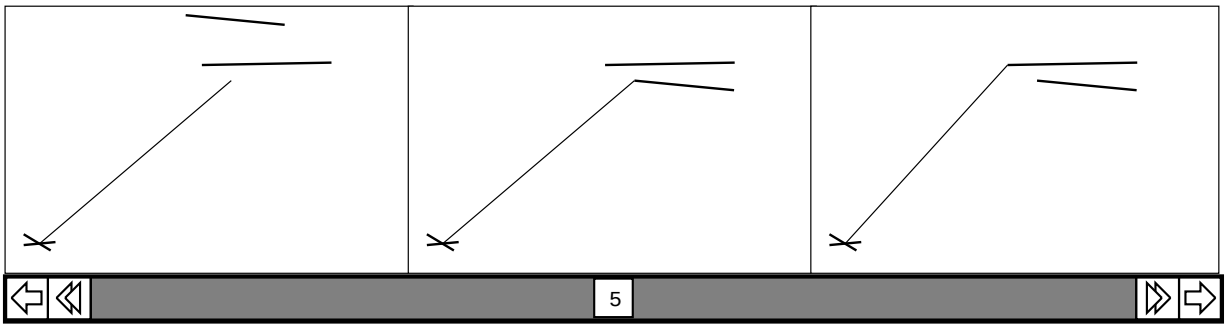

Figure 13: Example of a history board

convenient and effective to "communicate" through graphical objects, diagrams and tables.[20] Each approch has its own advantages and disadvantages: For example, a property form makes it easy to strech a line, but it makes it hard to move a line.

Since this concept is based of the property and relation functions it is general enough to be applicable in other domains than the graphic editor domain. The different styles of dialogs are based on the exploitation of the built-in property and relation functions so that generality of the concept can be reached. The scalability of the system is supported by an abstraction mechanism and the highlighting concept.

The only limitation of this concept is the incompleteness of the domain represented by the property and relation functions. A solution for this problem will be discussed in chapter 5.5.

5.4.2 Interpretation of Action Sequences

The mechanisms described in section 5.4.1 only provide simple *temporal* information about an action. This temporal information only deals with the last action so that connections were only made to the status of the editor before an action.

A major problem of modeling domains by property and relation functions is that they do not provide additional information that connects an action with previous actions, or properties and relations of objects in former system states. This task has to be performed by generalization functions. In order to provide a set of such functions that do not have to be adapted to each domain, the user has to enrich the provided information by additional interactions. This additional *temporal* information is used to support the search for loops and condition branches. Another aspect of adding temporal information by the user of a graphics editor is the reduction of the hypotheses space as described in 5.4.1.

Now, we want to discuss how the generalization function can be provided with more information by dialog-based interaction in order to enable a PbD system to generate a function schema. Dialogs, again, are the manipulation of objects, diagrams and tables by the user as well as the PbD system in an intertwined way.

[20] If one really wanted, one could also open a special dialog window and use question and answers exactly like for the apprentices.

Fig. 6 shows an example that needs many explanations regarding temporal information. For example, the generalization function has to explain that the third action (the rotation of line l_1) is due to its former connection to l_h. Additionally, it has to detect that line l_1 is rotated with the same angle as the helping line in the second action.

Of course, it makes sense to give the generalization function enough power to come up with such an explanation based on the previous system state. The generalization functions only have to check the parameters of the last action and the parameter of the property and relation functions of the previous system state. Nevertheless, we obtain the same problems as described in 5.2, because the amount of possible explanations can be exponential with respect to the length of the trace.

Additionally, if we only check the explanations that are constructed based on the previous system state, the function can possibly miss some. In the scene of fig. 6, for example, the generalization function cannot explain why line l_1 is moved in order to meet line l_h in the fourth action if it looks only one system state back.

One solution of the problem works similar to the forms idea in 5.4.1, but with an additional "communication channel". We have to give the user of the PbD system and the graphics editor the chance to freely access all history information. A simple list of all properties and actions provides the necessary information but the user may have problems with this unstructured approach. In order to provide the information in a more user-friendly way, we follow D. Kurlander [15] who has proposed a *history board*. We extend it in order to display the system's state in all past situations, all actions, and all explanations and properties.

The history board shows a customizable number (typically three or four) of shrunken scenes the user can browse through. When the demanded scene is found, the user simply clicks on it and the editor shows the magnified scene. The user is now able to operate inside the scene as described in section 5.4.1: He or she can open property and relation forms and transfer information by copy&dragging data from the past to the present scene.

Another approach to enrich the information of the PbD system is to provide the system with abilities to repeat the last action (or the last action of one class like "rotate" or "move"). This can simply be done by saving the last action and providing a pull down menu with the last actions the user was interested in. For example, in fig. 6, we can use the function "last rotate" in order to rotate lines l_1, l_2, and l_h in the third, sixth, and seventh action. This helps the user in two ways. First, the definition of the exact parameter of an action does not require much effort. The user has no problems to define, e.g. the rotation parameters again and again. Second, this information gives the generalization function helpful information in order to find loops in the action sequence.

Both approaches – history board and "repeat last action" – are general concepts that enable a user to give the PbD system additional hints and information about the user's intention. Again, both approaches are only based on the property and relation function provided by the PbD system.

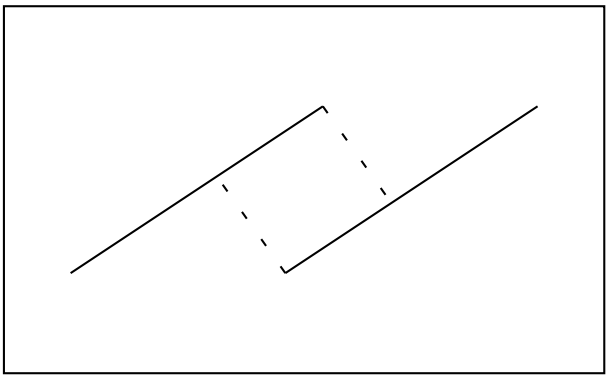

Figure 14: Two nearby-parallel lines

5.4.3 Dialogs during Generalization

Since the user wants to limit the amount of examples that is presented to a PbD system, the system has also to find the correct generalization even with insufficient information.

Without having special knowledge about the demonstrated task, this is not possible. For example, when a user defines a procedure to manipulate a polygon and uses a rectangle for his demonstration, it is quite obvious that a full generalization based on this example is impossible. Nevertheless, it would be useful to have a mechanism that does not need an additional example to conclude that the induced procedure is suitable for the manipulation of all kinds of polygons.

By introducing types like in [31] (or also the message types in RAP) the user can enrich the information so that the amount of needed examples can be reduced. To increase the confidence to the type generalization, the PbD system can ask the user

```
shall I apply the function to all convex polygons (y/n)?
```

as would be helpful in the construction of the "extend-3D" function described in [31]. So far, however, we don't have any experience how many such question should be provided for the complete editor, or whether one can define all of them a priory.

Another case where the user can help is to determine the number of times a loop has to be performed. The system could ask if it should perform a loop a fixed or variable number of times. In the case of a variable number of times the user supports the system by providing hints on the loop variable and/or the end condition of a *while* loop.

A similar problem is the detection of parameters and types of a function. In fig. 6 the intended function has 3 parameters. The first one is the meeting point p of two lines. The second parameter is a set of lines that have to be rotated. The third parameter is an action. The first rotation should be performed by the user in order to give the angle of the rotation. For example, the PbD system cannot decide, if the rotation is a parameter or only a simple action. By using question and answer mechanisms, the PbD system can be provided with this information.

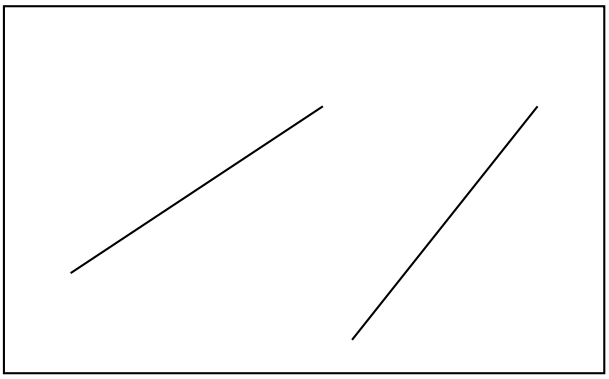

Figure 15: A first negative example for nearby-parallel lines

5.5 Learning New Property and Relation Functions

We saw in 5.2 that one of the problems of PbD is the limitation and incompleteness of the domain theory. Especially, if the user wants to use new concepts, he or she repeatedly needs to construct these properties during the demonstration of events instead of using a property that is only defined once.

In order to overcome this problem the user needs the ability to enrich the domain theory by adding new property and relation functions directly. Since the domain is modeled by property and relation functions expressed in a conventional programming language, the PbD approach can be used also for defining the new property and relation functions. The definition of the property and relation functions by using PbD solves parts of the incompleteness problem as described in [31]. Nevertheless, the approach is not able to generate *every* property and relation functions; only these that are derivable from the initial property and relation functions, and by the expressional power of demonstration and interaction mechanisms as provided by the PbD system.

Example

To point out the possibility of extending the set of property and relation functions by demonstration and dialog-based techniques (as discussed in the previous sections), we show how one can define the relation *nearby-parallel* (see fig. 14) of two lines.

When we use the metacalculator we can easily define an expression for the relation function *parallel*: Two lines are parallel if both have the same angles (with respect to some predefined coordinate system), or if the angles are different by 180 degrees. With some additional interaction within the PbD system, the user is able to name this function "parallel" and import it as a relation function in the PbD system.

In some applications it makes sense to extend the regular definition of parallel. We define two lines as *nearby-parallel* if they are parallel and one part of the first line can be orthogonally projected on the second line. Besides the decision if two lines are nearby-parallel, the relation also should give the distance between two nearby-parallel lines.

In order to demonstrate the new relation function we proceed as follows. First, we demon-

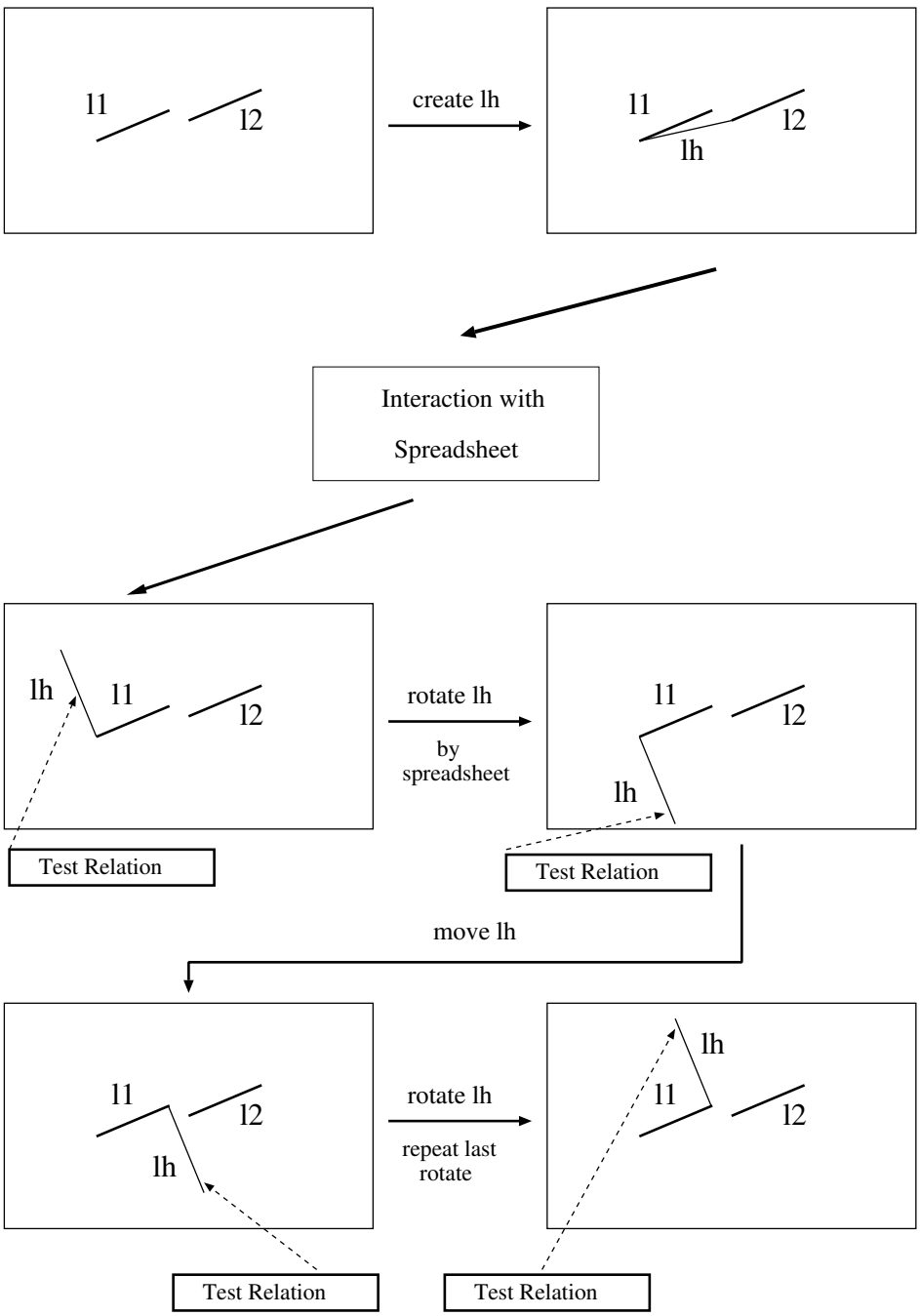

Figure 16: A second, more complex negative example

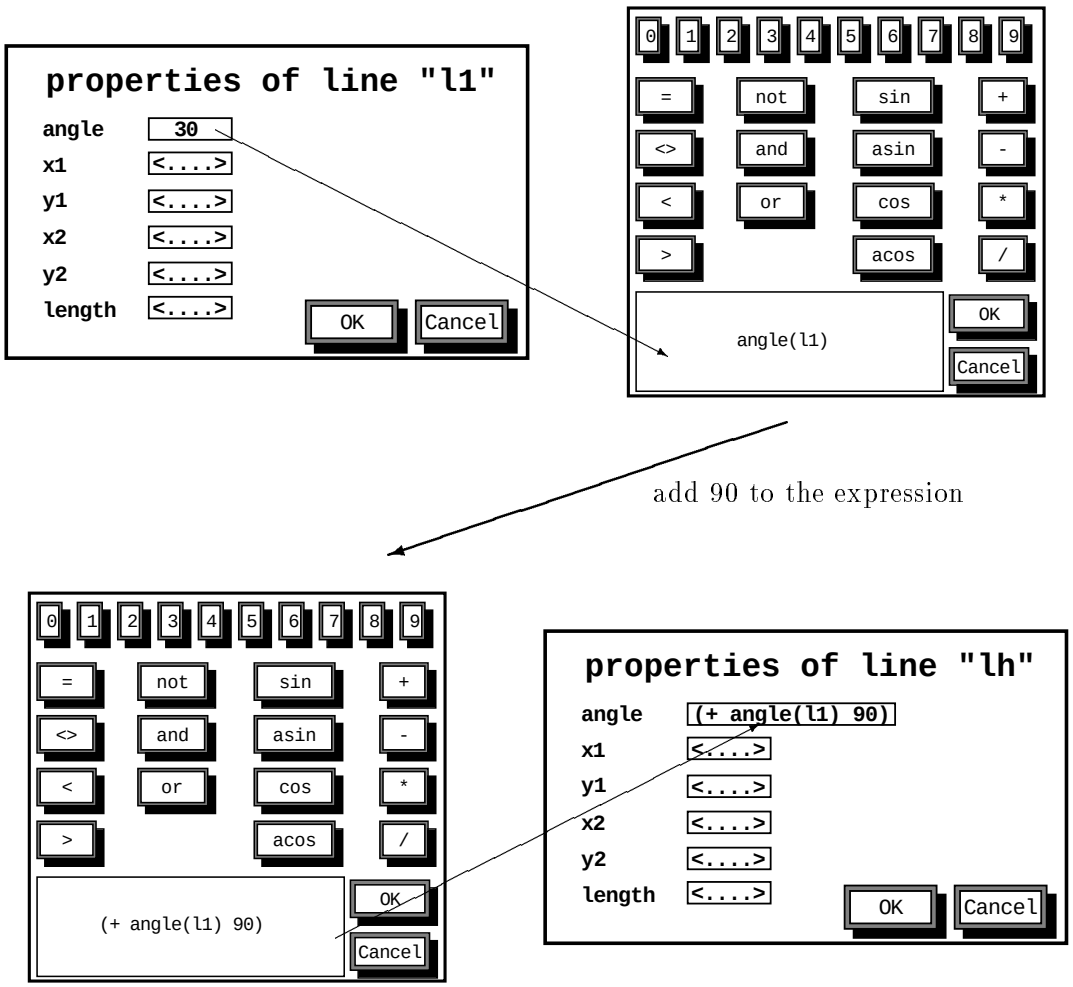

Figure 17: The interaction with the metacalculator

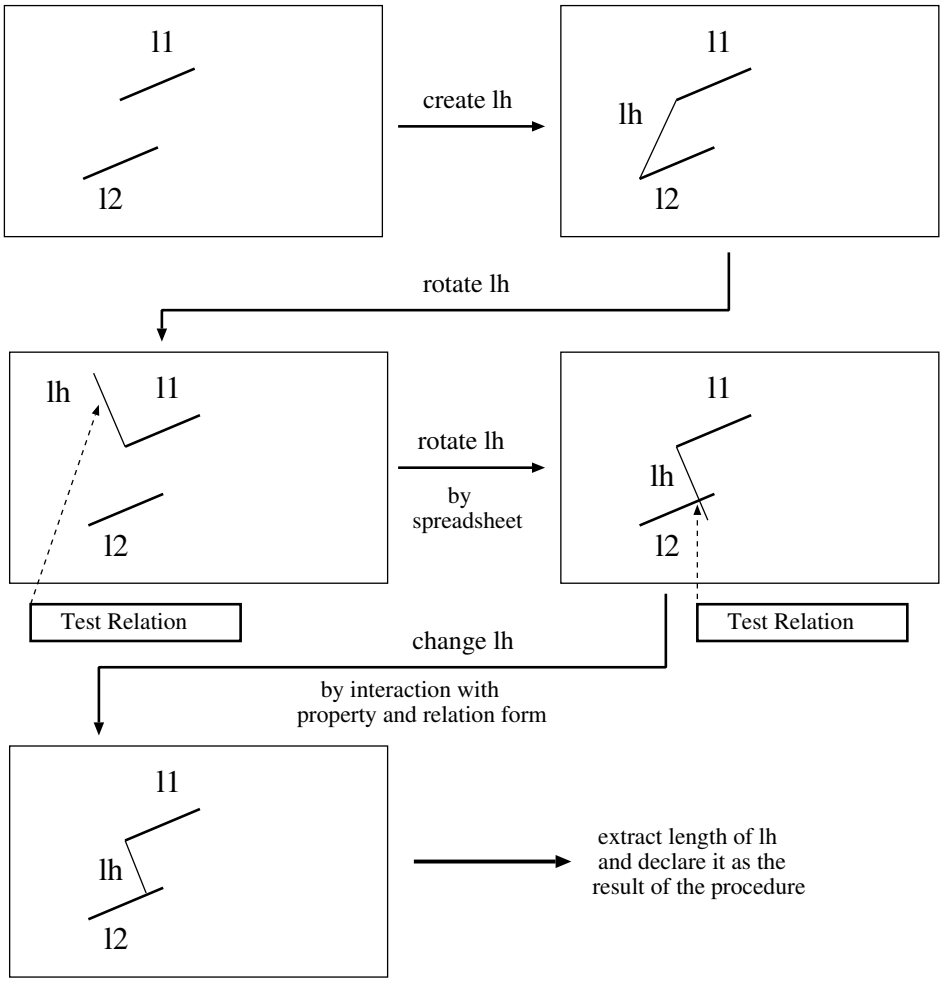

Figure 18: The first positive example for nearby-parallel lines

strate an negative example as shown in fig. 15. We draw two non-parallel lines l_1 and l_2 and declare them as parameters of the new relation function. Then we pop up the relation form of l_1, click on its neg–key (for negative example) and select the $parallel(l_1\ l_2\ NIL)$ relation. By doing this we "tell" the PbD system that we are only interested in the information that l_1 and l_2 are not parallel. In order to show that the example was a negative one, we have to stop the recording of the event by selection of "negative example" in the recording control in the PbD programming system.

Next, we demonstrate another negative example as shown in fig. 16. This time we start with the parallel lines l_1 and l_2 that do not meet the second condition of nearby-parallel. As in the first negative example, we declare l_1 and l_2 as parameters and pop up the relation form of l_1. Here we select the relation $parallel(l_1\ l_2\ T)$ and press "ok" in order to show the system[21] that

[21] Depending on how "intelligent" the underlying PbD system is, it comes to the conclusion by itself that the

we are only interested in the property "parallel" of both lines.

In the next step we introduce a helping line l_h that connects the beginning points of l_1 and l_2 and pop up the property forms of l_h and l_1. We then drag the property *angle* of l_h to the metacalculator (see fig. 17 where we add the value 90. Afterwards, we drag the expression $(+\ angle(l_1)\ 90)$ to the *angle* field of the property form of l_h. Automatically, the system rotates the line l_h around its beginning point. Additionally, the PbD system derives the complex action *rotate* that is dependent on properties of l_1, l_h and the constant value 90. Then we pop up the relation form of l_h, click on "neg" and select the relation *meets-on* $(l_h\ l_2\ NIL)$. Afterwards, we rotate l_h by 180 degrees by using the metacalculator and the property form of l_h and select again the relation *meets-on* $(l_h\ l_2\ NIL)$ in the relation form of l_h. Then we move the beginning point of l_h to the end point of l_1 and repeat the procedure with l_h at this point as at the beginning point of l_1. Since in each case l_h does not meet l_2 we have to select each time the negative relation of *meets-on*. We end the example with the deletion of l_h and the selection of "negative example" in the recording control of the PbD system.

After this demonstration, the PbD system recognizes that the property *parallel* is necessary in order to continue with the construction of the helping line l_h. Additionally, the PbD system concludes that after each movement of l_h the relation *meets-on* is tested and that the negative result of this test is the reason why the example is a negative one.

Next, we provide a positive example as in fig. 18 in order to add the lacking information of the relation function nearby-parallel. We start with two nearby-parallel lines l_1 and l_2 and continue as in the second negative example in fig. 16. This time we select the relation *meets-on* $(l_h\ l_2\ (x_m, y_m))$. Since we want to have the distance between l_1 and l_2 as the result parameter of the relation function nearby-parallel, we have to construct this result before we stop the recording.

With the same mechanism as in the previous example (shown in fig. 12), the end point of l_h can be moved to the meeting point of l_h and l_2. If we do this, the system automatically calculates the new length of l_h. Therefore, we can define the length of l_h after this step as the result parameter. We do this by clicking in the length field of the property form of l_h and selecting the "declare as result"–key in the PbD system. After we did this, we delete l_h and click on "positive", i.e. "tell" the PbD system that this example is a positive one.

We can now continue with providing some additional positive examples in order to test the reaction of the relation function in the remaining three cases. On the other hand the PbD system can also generate hypotheses that generate the first positive example for all four cases.

The result is a relation function "nearby-parallel" that we programmed by demonstration with only a few examples. This relation function has to be added to the built-in set of property and relation functions using the PbD system.

---

properties "parallel" or "not parallel" are criteria that make the first and the second example different. In this case the system asks the user if it came to the correct conclusion by showing property panels and inviting the user to interact as described previously.

5.6 Discussion

The previous sections have proposed ways to include dialogs into a demonstrational graphics editor. In addition to other approaches (like Kurlander's CHIMERA [15]), we do not only want to synthesize new functions but also extend the domain theory – expressed in form of property and relation functions – by the same mechanisms. Dialogs, as the method of interaction, are defined in a broader way than used for the apprentice systems described in the first part of this report. In addition to pure textual "communication" (like to ask `shall I apply the function to all convex polygons (y/n)?`; see section 5.4.3), we provide panels, property and relation sheets, metacalculators, and a history board. Both, user as well as the PbD system, can initiate a dialog by popping up one of those elements, manipulating them and "ask" the other one to proceed. Through those means, explanations or requirements of the user can be made "visible" to the PbD system and help it to interpret a situation, to generalize, and evetually synthesize a new function.

The major gain is that the PbD system can detect the user intention more easily when interpreting given examples. In the same way, the PbD system can tackle the problem of an incomplete domain theory (as addressed in [31]). If necessary, the user can demonstrate new property functions to enable the PbD system the interpretation of examples also for novel concepts. Espexially the possibility to handle positive *and* negative examples facilitates that job very much.

The economical advantages are quite the same as in the case of office automation software: A larger user acceptance because of the comfortable and powerful generalizing macro facilities, and a faster, more robust, and, therefore, cheaper development of such software. Moreover, the method can be seen as a kind of *software bootstrapping* approach. It can be applied to develop extendable workstation tools by incrementally building them up from a small set of manually coded functions.

Three important problems that are not solved so far include, first, the initial selection of a first domain theory. I.e. can we find a set of property functions that is "powerful" enough to build all potentially required extensions on top of it? How can we do that in a systematic way?

The second problem is the selection of a function scheme that shall be the template for a newly synthesized function. Shall the user select one from a list of predefined schemes? Can he propose new ones? How far is his choice constrained by the PbD systems's interpretation of the action sequence? How can we deal with recursive schemes? Other open questions, not addressed in this report, deal with the problems of correctness of the synthesized code. How far can we "trust" it? Is it easy to find and correct mistakes from incorrect functions? Can one remedy those functions? Does the PbD system or the user initiate such a "correction" step? Can the PbD system correct itself automatically if failures occured?

The third problem, concerning the interface to the end user, is how understable one can make the questions he is asked. How many of them have to be provided? How clear can they be structured? How can one ensure that the user won't become bored and just stops his activity?

6 Conclusion

In this report we discussed possibilities to involve users of learning apprentices as well as programming-by-demonstration systems in the performed induction and synthesis processes. They take part in those steps via *dialogs* that can be initiated by themselves, but also by the programs.

We made proposals on how to include the dialog-based learning (DBL) in the calendar apprentices CAP and CAP II, in the room reservation apprentice RAP, and in a demonstrational graphics editor. The mentioned systems are partly used in daily work (CAP), partly ready prototypes (CAP II, RAP), and partly still being under implementation (graphics editor developed with GARNET). The possible application of DBL was outlined for the

- Acquisition of fact knowledge about people, environment and events in CAP. This data is necessary for the automated learning of rules to give the user advice when setting up a meeting.
- Acquisition of alias list about peoples' names and email addresses in CAP II.
- Acquisition of the workflow model that describes the task of reserving a room via email in RAP. The model is represented as a finite state machine which drives – after a while – the reservation process automatically.
- Asking for explanations of user actions in a demonstrational graphics editor. With the explanations the PbD system can solve ambiguities on its way to synthesize new functions and when enhancing the domain theory.
- "Telling" the demonstrational graphics editor "invisible" constraints between objects via property and relation sheets, metacalculators, and a history board.

We used those applications to support our claim that – through dialog-based support – extendable and customizable software systems can provide

- **faster applicability:**
  See the user's "discussion" of newly learned rules (section 3.2) with the CAP agent that enables it, after very few events, to increase their usefulness (i.e. the confidence factor). Another example is the "discussion" of explanations with the PbD graphics editor (section 5.4) that can reduce search spaces drastically.
- **more complex functionality:**
  See the incremental acquisition of a workflow concept like that of RAP (section 4.4), and the acquisition of new domain theory concepts in the PbD graphics editor (section 5.5) that allow the user to synthesize arbitrarily complicated new functions.
- **success in meeting the user's intention:**
  Again, the "discussion" of learned rules in CAP and of explanations in the PbD graphics editor may serve as examples. Another one is the possibility to use helping lines, property forms etc. to explicitly show the graphics editor what one wants.

That could never be achieved, in our opinion, from software systems without DBL, if acceptable development costs were a realization constraint. Nevertheless, DBL should be seen as a *supplement* to other learning techniques rather than a substitute.

The possible advantages of interactive software systems with DBL capabilities – for both, end users as well as software developers (see the discussion in the sections 4.5 and 5.6) – seem obvious. However, nobody has proved yet that this kind of software systems will really be accepted as expected. Therefore, one major effort of our future work will be experimentation with the prototypes and field tests with internal, and as soon as possible, also external users.

7 Future Work

Two long-term goals are the reason for our research in dialog-based learning: A *software secretarial agent* that can assist with a wide variety of tasks, and that can be field-customized to individual users, and *user-extendable tools* (not only for workstations, also for CIM development etc.) that can be customized by explicit PbD mechanism. From that overall objectives the planned research steps are derived:

- RAP-ADMIN: the room apprentice on the room administrators' sites. Analogous to RAP, RAP-ADMIN shall learn to act on behalf of the room-administrating person and automatically answer on reservation questions. The problem here is that this apprentice must have access to the bookkeeping lists and perform some scheduling operations. Can this also be acquired, or demonstrated, or have we to pre-program it?

- As discussed in section 3.2, we soon want to add a DBL interface to the calendar apprentice CAP that enables the user to "review" interactively the learned rules. That mechanism is expected to drastically shorten the time one has to wait until a new rule can confidently be applied. The DBL interface shall especially support the difficult learning of rules for date and time advice.

- If the RAP experiment is successful, we want to apply the ideas to the "re-learning" of workflow and negotiation in CAP II. This would allow us to compare the learned version with the programmed one and analyze the advantages and limitations of our DBL approach on a "big" office automation problem.

- The combination of DBL and PbD systems shall not only be studied in the particular graphics editor domain. It is a step towards answering the question whether PbD can be used as a widely applicable *paradigm* that is a real alternative to "usual hand-coding", at least in some domains. Therefore, software development for robots, machining centers, household appliances, communication devices, consumer electronics etc. shall also be considered.

- One major research effort has also to be the study of suitable and accepted interfaces to the learning interface agents and customizable tools. This addresses in particular the issue

of what questions the user should be asked and how they should look like. Maybe some "Wizard-of-Oz" experiments could give a good answer to that before investing too much in further implementations.

Besides those DBL-oriented research goals we plan to explore different enhancements and new features, like the connection of apprentices to external knowledge sources (dictionaries, personal databases) and the provision of electronic forms (EFORMS) as an alternative interface.[22] Research in automated learning has to tackle questions on how to deal with irregularities, how to handle large feature spaces etc. A large portion will be experimentation and, at the last, we have to combine all our prototypes and single tools towards a software secretary like office automation system.

Acknowledgements

The authors gratefully acknowledge various fruitful discussions and valuable comments by David Kosbie, Tom Mitchell, Ryusuke Masuoka, David Zabowski, Dan Ferrell, Dayne Freitag, Lukas Rücker and Laurel Simmons. Furthermore, David and Dayne assited a lot in tuning our "Germlish" to readable English. We also want to thank our "Munich home base" – Sybille Rosenmüller, Gisbert Lawitzky, Wolfgang Rencken, and Gerd Sandweg – who handled the necessary "remote bureaucracy jobs" always fast and reliably.

[22] More about these enhancements can be found in the future work section of [1].

References

[1] S. Bocionek and T. Mitchell. CAP II: Making the Calendar Apprentice an Agent. Technical Report CMU-CS-93-to-appear, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1993.

[2] 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.

[3] J.G. Carbonell, R.S. Michalski, and T.M. Mitchell. Understanding the nature of learning: Issues and research directions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach*, chapter 1, pages 3 – 23. Tioga Publishing Comp., Palo Alto, CA, 1983.

[4] D. Chapman. *Vision, Instruction and Action.* MIT Press, Cambridge, MA, 1991.

[5] A. Cypher. EAGER: Programming repetitive tasks by example. In *CHI '91*, pages 33 – 39. ACM, 1991.

[6] R. Davis. TEIRESIAS: Applications of meta-level knowledge. In R. Davis and D. Lenat, editors, *Knowledge-based systems in artificial intelligence.* McGraw-Hill, New York, 1982.

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

[8] 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.

[9] H.L. Dreyfus. *What Computers Still Can't Do.* MIT Press, Cambridge, MA, 1992.

[10] N. Haas and G.G. Hendrix. Learning by being told: Acquiring knowledge for information management. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 13, pages 405 – 427. Tioga Publishing Comp., Palo Alto, CA, 1983.

[11] 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.

[12] Y. Kodratoff and R. Michalski. *Machine Learning, An Artificial Intelligence Approach*, volume III. Tioga Publishing Comp., San Mateo, CA, 1990.

[13] D.S. Kosbie and B.A. Myers. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, chapter 22. 1993.

[14] 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.

[15] D. Kurlander and S. Feiner. A history-based macro by example system. In *UIST '92 Proceedings*. UIST '92, November 1992.

[16] H. Lieberman. An example based environment for beginning programmers. *Instructional Science*, 14:277 – 292, 1986.

[17] D.L. Maulsby, I.H. Witten, and K.A. Kittlitz. METAMOUSE: Specifying graphical procedures by example. *ACM Computer Graphics*, 23(3):127 – 136, 1989.

[18] D.L. Maulsby, I.H. Witten, K.A. Kittlitz, and V.G. Franceschin. Inferring graphical procedures: The compleat METAMOUSE. Research Report 90/388/12, University of Calgary, Department of Computer Science, Calgary, Alberta, Canada, May 1990.

[19] 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.

[20] 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.

[21] R.S. Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine Learning, An Artificial Intelligence Approach*. Tioga Publishing Comp., Palo Alto, CA, 1983.

[22] R.S. Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine Learning, An Artificial Intelligence Approach*, volume II. Morgan Kaufman Publishers, Inc., Los Altos, CA, 1986.

[23] R.S. Michalski and Y. Kodratoff. Research in machine learning: Recent progress, classification, and future directions. In Y. Kodratoff and R. Michalski, editors, *Machine Learning III, An Artificial Intelligence Approach*, chapter 1, pages 3 – 30. Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[24] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: a unified view. *Machine Learning*, 1(1):47–80, 1986.

[25] 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.

[26] D.A. Mostow. Machine transformation of advice into a heuristic search procedure. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial*

Intelligence Approach, chapter 12, pages 367 – 403. Tioga Publishing Comp., Palo Alto, CA, 1983.

[27] B.A. Myers. Creating user interfaces by demonstration. Technical Report CSRI-196, University of Toronto, Computer Systems Research Institute, May 1987.

[28] 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.

[29] M.D. Rychener. The instructible production system: A retrospective analysis. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 14, pages 429 – 459. Tioga Publishing Comp., Palo Alto, CA, 1983.

[30] C. Sammut and R.B. Banerij. Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning II, An Artificial Intelligence Approach*, chapter 7, pages 167 – 191. Morgan Kaufman Publishers, Inc., Los Altos, CA, 1986.

[31] M. Sassin and S. Bocionek. Programming by Demonstration (PbD): The Basis for Auto-Customizable Workstation Software. In A.H. Rubenstein and H. Schwärtzel, editors, *Intelligent Workstations for Professionals*, pages 199 – 220. Springer-Verlag, 1993.

[32] R. Schank and A. Kass. Explanations, machine learning, and creativity. In Y. Kodratoff and R. Michalski, editors, *Machine Learning III, An Artificial Intelligence Approach*, chapter 2, pages 31 – 48. Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[33] 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.

[34] E.Y. Shapiro. Inductive Inference of Theories From Facts. Technical Report 192, Yale University, Department of Computer Science, February 1981.

[35] 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.

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

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

[38] K. VanLehn. Learning one subprocedure per lesson. *Artificial Intelligence*, 31:1–40, 1987.

[39] D.C. Wilkins. Knowledge Base Refinement As Improving An Incorrect And Incomplete Domain Theory. Technical Report UIUCDCS-R-90-1585, Department of Computer Science, University of Illinois, Urbana, Il, April 1990.

[40] D.C. Wilkins, W.J. Clancey, and B.G. Buchanan. Knowledge base refinement by monitoring abstract control knowledge. *International Journal of Man-Machine Studies*, 27:281 – 293, 1987.