Project No. 82

REPRESENTATION ASPECTS OF KNOWLEDGE-BASED OFFICE SYSTEMS

K. Van Marcke, V. Jonckers and W. Daelemans
A.I.-LAB, Vrije Universiteit Brussel
Pleinlaan 2 Building K
1050 Brussels, Belgium

ABSTRACT

The main goal of ESPRIT project 82 is to build an intelligent office workstation (IWS). A knowledge-based approach was chosen to overcome the complexity resulting from the open-endedness of the office domain and the dynamic characteristics of the office environment. The knowledge representation system KRS is used to represent knowledge in the office domain. On the one hand, domain knowledge involves static domain concepts including concrete concepts such as persons and communication-means and more abstract concepts like organisations and roles. On the other hand, it involves a range of office activities which operate on these static domain concepts. In general, two categories of activities can be distinguished: primitive actions which directly operate on the office environment like 'send letter' or 'archive document' and higher level conceptual tasks like 'plan a trip' or 'order a book'. The execution of actions and the translation of tasks into actions must be controlled within the IWS. For this purpose, special office formalisms such as agendas, monitors and a unification mechanism were designed and implemented on top of KRS. The paper presents these office formalisms and illustrates their functionality with the description of a prototype knowledge-based office assistant and with some worked out examples of typical office-tasks.

1. INTRODUCTION

IWS (ESPRIT Project 82) is a project in the domain of intelligent office applications. The partners involved in the project are Bull (France), VUB (Belgium), KUN and OCE (the Netherlands) and CRC (Greece). The goal of the project is to build an intelligent cooperative personal workstation that assists the office-worker in the office tasks he is playing a role in.

An important aspect is the representation of office knowledge. In order to let the workstation intervene in an intelligent and intelligible way, it must know about the office-procedures that are being followed and about their states, goals and reasons. To this end, the system must be knowledgeable about the functioning of offices in general (the generic model) and about the organisation of the user in particular. At each of these levels, three types of knowledge can be distinguished: *tool-knowledge* (how to use electronic tools such as text-formatters, printers, files, telefaxes, etc.), *domain knowledge* (in the generic model: knowledge about hierarchies, approvals and signatures, communication means, deadlines, book-years, vacations etc.; in the particular model: instances of the generic concepts, organisation structure, the people that work in the organisation and their roles, goals of the organisation and its function in society etc.) and *office procedure knowledge* (the knowledge needed to interact in the office environment:

knowledge about office-procedures, tasks and activities, problems, problem-solving, planning and waiting.

The representation tool system KRS (Steels, 1986; Van Marcke, 1987) was chosen for the representation of these different types of knowledge. KRS was developed at the VUB AI-Lab within the ESPRIT project P440 (Message-Passing Architectures and Description Systems). It is based on earlier work on Representation Language Languages (Greiner, 1980). KRS is basically a flexible object-oriented language in which multiple knowledge retrieval architectures can be explicitly modeled and used. The advantage of modeling different representation formalisms explicitly in the same language is that it allows knowledge represented in different styles to cooperate. A more pragmatic advantage of using KRS is that it is implemented in ZetaLisp in such a way that it can be easily ported to other dialects, and that it runs efficiently. KRS has been successfully ported to InterLisp, Common Lisp and Le-Lisp. Porting to Le-Lisp has been done within the context of the IWS project by people at Bull, because Le-Lisp was chosen as the language to be used in the project.

The task of the VUB in the project is to design and develop knowledge representation formalisms in KRS. Those must facilitate the task of representing office knowledge (carried out by Bull, see the companion paper to this one: Ader, 1987). The present paper describes the current state of this process. More specifically, it describes three different formalisms which are currently being developed and used in our laboratory. Some of them are also used in the implementation of the activity-manager at Bull (Tueni et al., 1987). The first is the representation of priority-agendas. A priority-agenda is an agenda on which items can be placed together with a priority. The item with the highest priority on the agenda is always removed first. Multiple priority-agendas can be used to control execution of tasks and actions in a flexible and modular way. A second formalism which has been developed in KRS is a monitor formalism. Monitors allow to automatically start some activity the moment some event occurs. In an office environment, most actions are typically invoked as a reaction to some event. The third formalism enables the representation of features and the unification of feature-sets, which is among other things useful to decide between alternative solutions.

The paper first gives an overview of the aims and achievements of KRS. It is not our intention to give a detailed description of the mechanisms of the language, but rather to illustrate why it is a suitable tool for our purposes. Secondly, the three special-purpose formalisms introduced are illustrated by means of an experimental architecture simulating an office environment. This experimental architecture is being developed at VUB to test and to experiment with those formalisms. Although this office environment will not be part of the final specifications of the project, it is sophisticated enough to be a non-trivial test-bank for the formalisms we propose.

2. KRS

2.1. Philosophy

KRS is a representation tool system developed at the VUB AI-Lab. It is partly inspired by earlier work in Representation Language Languages and in particular by the language ARLO (Haase, 1986). KRS differs from popular representation tool systems like KEE and ART in that it does not provide a fixed set of formalisms. Instead it provides a language, called the concept-language, in which new or existing formalisms can be easily modeled and used by the user. It is essentially an object-oriented language featuring single inheritance, lazy construction, caching, consistency maintenance, meta-representations and explicit representation of reference.

Multiple inheritance has grown into a heavily debated controversy in the object-oriented systems community. KRS has chosen for a single inheritance mechanism with the capability to build more complicated architectures explicitly into the language when they are needed (Van Marcke, 1987). KRS objects are constructed in a lazy way, i.e. not when they are defined but when they are first used. This leads to more economic applications and more elegant definitions, especially when cross-pointing structures are involved. Caching is a technique to store computed values for re-use. This may lead (depending on the application) to serious performance upgrades. Together with caching, a consistency maintenance mechanism is needed to make sure that there remain no cached values depending on something that has already changed. All KRS objects also have a unique meta-interpreter object that contains all information about the object itself. Those meta-features are explicitly accessible and can be overridden by the KRS-user (Maes, 1987). Finally, the *referent* relation explicitly states what an object is about. This has led among others to a smooth and sound interface to LISP. A more thorough description of KRS can be found in (Steels, 1986).

KRS was originally implemented on a Symbolics LISP machine in ZetaLisp. An earlier version has been ported to INTERLISP to run on a Xerox LISP Machine. The latest version has been ported to Common Lisp by ourselves to run on for example a Sun work station. It has also been ported to Le-Lisp by Michel Tueni of Bull to run on Sun, Vax and the Metaviseur. The implementation in Le-Lisp was made in the context of this project.

2.2. The Concept-Language

The fundamental building block in the concept-language is the *concept*. A concept is a frame-like structure, grouping a set of subjects. Each subject is a named association between the concept and some other concept, which we call the filler of the subject. This way concepts are connected in a labeled network. We will describe a concept with its name (if it has one) possibly followed by a list of its subjects, like in the following example:

```
JOHN
  A PERSON
    AGE TWENTY-FIVE
    FRIEND A PERSON
             AGE (>> age)
```

The subjects are described with their name and a description of their filler. The description of this filler can be the description of this concept, or a reference to the filler of another subject. In the preceding example, the filler of the age-subject of the filler of the friend-subject of the concept with name John is described by a reference to the filler of the age-subject of that same concept ("(>> age)"). The example therefore describes a concept named John, with an age-subject with filler the concept named Twenty-Five, and with a friend-subject with filler a concept with a new age-subject with filler also the concept named Twenty-Five.

The descriptions of the concept John and of the filler of its friend-subject contain both an expression "A PERSON". This expression classifies the concepts in a strict type-tree, which is used for a prototype-style (Lieberman, 1986) inheritance. We say that the concept Person is the type of both concepts. A concept inherits all subjects from its type except those which it overrides.

The KRS concept-system is a representation language language, i.e. a language to describe representation languages. Most of the static knowledge structures are most naturally expressed directly in the concept-language. When more complicated or more dynamical architectures are needed, they can be built explicitly in the language. KRS provides libraries to facilitate this construction. Some of those libraries have already been developed and/or experimented with, such as a production-rule library, a logic-library, a cliche-library, etc. (Steels et al., 1987).

3. A PROTOTYPE OFFICE MANAGER

3.1. A General Office Model

To investigate which formalisms can be useful in the implementation of an intelligent office system and to develop and experiment with such formalisms we have developed a prototype office manager which implements a very general model of an office environment. This section first sketches this general office model and introduces the basic concepts used in its implementation. The need for dedicated formalisms to complete this implementation is briefly discussed at the end of this section. In the next section three formalisms are discussed in more detail.

In the office domain static and active knowledge components can be distinguished. Static office knowledge involves knowledge about concrete and abstract concepts in the office domain, e.g. persons, communication-means, organisations, roles, etc. Such office concepts can be modeled straightforwardly as KRS-concepts. In this approach, the KRS-concepts fulfill the role of frame-style representation structures. Modeling office concepts as KRS-concepts therefore holds the advantages commonly attributed to frame-based representation languages (Fikes 1985). Frames capture the way experts typically think about much of their knowledge. They can describe different types of domain objects, they can represent the useful relations, and they support a kind of 'definition by specialisation' which is in most domains a natural thing to do. Below we show some simple examples (in a simplified notation) of such KRS-concepts modeling a prototypical person and a specific person, respectively.

```
PERSON
  A STATIC-OFFICE-CONCEPT
    NAME a person-name
    ADDRESS an address
    ELECTRONIC-MAIL-ADDRESS a mail-address

KRIS
  A PERSON
    NAME  A PERSON-NAME
            LAST-NAME [string "Van Marcke"]
            FIRST-NAME [string "Kris"]
    ELECTRONIC-MAIL-ADDRESS [mail-address krisvm@arti.vub.uucp]
```

Complementary, office activities which operate on these static domain concepts can be viewed as active knowledge components. Furthermore, activities like 'send a letter' or 'archive document' which correspond to concrete actions in the office domain can be distinguished from activities like 'plan a trip' or 'order a book' which are more conceptual tasks. In fact, the latter require a combination of primitive actions for their execution. Below we sketch a very general model of active office knowledge. A complex task gradually decomposes into more primitive subtasks. In the end, each primitive subtask corresponds straightforwardly to a single concrete action.

In the remainder of the text, the term *action* is reserved for a concrete, executable action in the office world while the term *task* is used to denote conceptual entities of activity. In our terminology, actions are *executed* while tasks are *handled*.

3.2. Tasks and Actions

A KRS-concept named *action* is introduced to serve as ancestor node for all KRS-concepts which model an action. As usual, subjects associate the necessary information with each action concept. In particular, the procedure which can be called to execute the action is associated with each action concept via the *execute-procedure* subject. An example is shown below.

```
ACTION
  AN ACTIVE-OFFICE-CONCEPT
     EXECUTE-PROCEDURE a procedure

SEND-ELECTRONIC-MAIL-ACTION
  AN ACTION
     SENDER a person
     RECEIVER a person
     MESSAGE a message
     EXECUTE-PROCEDURE [procedure (send-mail ...)]
```

A KRS-concept named *task* is introduced to serve as ancestor node for each KRS-concept which models a task. The task distribute-message discribed further on is an example for this concept. A task contains a script which is in general an analysis of the tasks in terms of more simple tasks. Such a script, called a task-handling-script, determines how the task is handled, either in terms of its subtasks or in terms of an action to be executed. A task-handling-script has a handle-procedure subject, typically filled by a lisp-procedure, that implements the functionality of the script. Libraries of general-purpose scripts can be provided and applied to handle more specific tasks.

There are two overall subcategories of task-handling-scripts. *Primitive-task-scripts* translate the task in exactly one action to be executed. We call tasks with a primitive-task-script *primitive-tasks*. *Composed-task-scripts* match a predefined combination of more basic tasks. Typical examples of composed-task-scripts are do-all-tasks, do-all-tasks-sequentially, do-some-tasks, do-best-task, etc. They play the same role as control abstractions in programming languages.

To handle a primitive task, the corresponding action must be executed. Therefore, an action is associated with each primitive-task-script via the *handle-action* subject. To handle a composed task, subtasks must be handled as determined by the semantics of the composed task. To handle a *do-all-tasks* for example, each task enumerated in the *tasks* subject of this composed-task-script must be handled.

```
TASK-HANDLING-SCRIPT
  AN OFFICE-CONCEPT
     HANDLE-PROCEDURE a procedure

PRIMITIVE-TASK-SCRIPT                          COMPOSED-TASK-SCRIPT
  A TASK-HANDLING-SCRIPT                         A TASK-HANDLING-SCRIPT
    HANDLE-ACTION an action
    HANDLE-PROCEDURE                           DO-ALL-TASKS
      [procedure (>> execute handle-action)]     A COMPOSED-TASK-SCRIPT
                                                   TASKS  a task-list
                                                   HANDLE-PROCEDURE [procedure (mapcar ...)]
```

The example below shows how a simple task is defined using a primitive-task-script.

```
TASK
  AN OFFICE-CONCEPT
     SCRIPT a composed-task

SEND-MESSAGE
  A TASK
    FROM a person
    TO a person
    MESSAGE a message
    SCRIPT A PRIMITIVE-TASK-SCRIPT
             HANDLE-ACTION A SEND-ELECTRONIC-MAIL-ACTION
                           SENDER (>> from)
                           RECEIVER (>> to)
                           MESSAGE (>> message)
```

The next example illustrates how a complex task such as distributing a message to a set of persons is described on a very abstract level, i.e. in terms of the message to send, the person who sends the message and the list of persons who have to receive the message. The script role holds the appropriate composed-task-script. Notice that the programming cliche *transform* is used to obtain the sequence of individual send-electronic-mail tasks from the information given with the distribute-message-task.

```
DISTRIBUTE-MESSAGE
  A TASK
    FROM a person
    DISTRIBUTE-TO  a person-list
    MESSAGE a message
    SCRIPT A DO-ALL-TASKS
             TASKS (A TRANSFORM
                      TRANSFORM-OVER (>> distribute-to)
                      TRANSFORM-WITH ?one-person
                      TRANSFORMER A SEND-MESSAGE
                                    FROM (>> from)
                                    TO ?one-person
                                    MESSAGE (>> message)
```

It may be necessary to emphasise at this point the importance of the distinction between primitive-tasks and actions. It is true that every primitive-task translates straightforwardly to an executable action. But a primitive task is no more than a conceptual entity while an action

really operates on the environment. Therefore, dealing with failures, recovery after machine crashes, and other persistency problems are different for both kind of concepts.

From the very general model sketched so far, the need for mechanisms which control the decomposition of tasks into more primitive subtasks and the execution of the resulting actions automatically emerges. The simple examples given create the impression that modeling complex office tasks corresponds to writing a kind of high level procedures in a specialised and powerful language. But the execution of these procedures can not be controlled by a fixed interpretation mechanism. A much more flexible interpretation process is necessary to deal with the typical characteristics of an office environment such as:
- The competition for resources between different tasks and subtasks which can be reflected in differences in priority.
- Interference of externally induced changes in the office environment with the execution of tasks.
- Interference between execution of different tasks.
- The existence of alternative ways for dealing with the same problem.

The following section introduces three special purpose formalisms which are useful in the implementation of an office manager. First, *priority-agendas* are introduced and their usage to control the execution of tasks and actions with different priorities in a flexible and modular way is illustrated. Second, *monitors* are introduced and it is illustrated how they can be used to coordinate task handling. Finally, a *unification mechanism* on feature sets is proposed which can be used to deal with choices between alternatives.

4. SPECIAL-PURPOSE OFFICE FORMALISMS IN KRS

4.1. Priority Agendas

4.1.1. Basic Idea

A priority-agenda is basically a queue on which items can be stored together with a *priority-level*. The selection of the number of priority-levels is arbitrary. Upon a *first-element* request, the agenda returns the items in order of decreasing priority. Objects with the same priority are returned on a first in first out basis. In addition, the priority-levels of the objects on the agenda can be *upgraded*. For this purpose, an *upgrade-step* and a *maximum-upgrade-level* is associated with each priority agenda. Regular upgrades can for example be used to avoid that tasks with low priority-levels stay on the agenda for ever.

```
PRIORITY-AGENDA
     AGENDA a mutating-list
     UPGRADE-STEP a number
     MAXIMUM-UPGRADE-LEVEL  a number
```

Priority-agendas as such are useful for many purposes. In particular, we illustrate in the following subsection how priority-agendas are used in the implementation of the prototype office manager which serves as test-bed for the development of special-purpose office formalisms.

4.1.2. The Administrative Coordinating Manager

The prototype office-manager is represented as a KRS-concept named Administrative-Coordinating-Manager (ACM). An ACM has three important components: a task-agenda, an action-agenda, and a monitor-manager. The task-agenda component is responsible for handling tasks in the office-environment, i.e. for the translation of tasks into a number of subtasks to be handled. The action-agenda component is responsible for controlling all operations on or within the office-environment, i.e. for executing actions. The monitor-manager component is responsible for feeling changes in the office-environment and for activating tasks and actions that are waiting for such events to happen. This component is described in more detail in the following section when monitors are discussed.

Both the task-agenda and the action-agenda are specialisations of the priority-agenda discussed earlier. All items on the task-agenda are tasks and all items on the action-agenda are actions. Furthermore, each agenda has associated with it an independent LISP-process that infinitely selects and 'treats' the first-element of the agenda. In the case of the action-agenda this simply means that the actions on the agenda are continuously executed. In the case of the task-agenda this means that the tasks on the agenda are handled which implies that subtasks are pushed on the task-agenda or that actions are pushed on the action-agenda as specified for the task involved. In this way, a few simple components are combined into a powerful yet flexible and modular whole.

```
ADMINISTRATIVE-COORDINATING-MANAGER
  AN OFFICE-CONCEPT
     TASK-AGENDA a priority-agenda
     ACTION-AGENDA a priority-agenda
     MONITOR-MANAGER a monitor-manager
```

Using priority-agendas to run tasks and actions holds several advantages over a fixed interpretation process for tasks and actions. In an office environment multiple activities are running simultaneously. The use of a priority-agenda makes it for example possible to handle urgent new tasks before completion of older, possibly already partially handled tasks. As explained earlier, to handle a complex task, subtasks are pushed on the agenda (as specified in the task's script). Therefore, another urgent task can be handled before a cumbersome but maybe not so important task is completely finished since the ACM will detect this newer and more urgent task. In combination with the monitor-manager discussed in the following subsection it becomes also possible to explicitly suspend the execution of tasks and to reactivate them automatically when appropriate.

Using two separate agendas for tasks and actions is decided upon because it further enlightens the conceptual difference between tasks and actions. In addition, using different agendas implies using different concurrent processes to execute actions and to handle tasks, respectively. This may turn out to be an advantage when for example a high-priority action is taking up much resources since part of the resources will always remain available for running the task-agenda and vice versa.

4.2. Monitors

4.2.1. Basic Idea

A *monitor* is a KRS concept which performs an action when it senses a change in the concept it monitors (the filler of its *monotoring* subject) and only if its *fire-when* subject is true at that moment. Monitors are connected to the dependency network used in the consistency maintenance system of KRS and can detect mutations to concepts that way. The *fire-action* subject of a monitor contains a form to be evaluated whenever a change is monitored. The monitor starts monitoring when the referent of its *activate* subject is computed. A monitor is deactivated when its *deactivate* subject becomes true. Once deactivated, a monitor can never fire again.

```
MONITOR
   MONITORING  a concept
   FIRE-ACTION a form
   FIRE-WHEN   a boolean
   DEACTIVATE  a boolean
   ACTIVATE a switch
```

Monitors are used to implement the monitor-manager component of the ACM. This component manages the resumption of the execution of primitive and composite tasks that were waiting for events to happen. Central in the operation of the monitor-manager is the concept of a *delay* which is based on the activation of a monitor.

4.2.2. Delays

A *delay* is a concept which is used to suspend execution of tasks until something happens.

```
DELAY
  ACTIVATE
  EVENT  some mutating concept
  DELAY-UNTIL a form
  DELAY-WHAT a task
```

The delay concept has subjects *event* (the concept which represents the event that is monitored for changes); *delay-what* (a primitive or composed task which is to be suspended until something happens); *delay-until* (a condition to be checked when the monitored event changes; when this condition becomes true the delay is ended and the suspended task is resumed) and *activate* (to start the delay: the *delay-what* is suspended, and a monitor is installed and activated monitoring *event*, firing when *delay-until* becomes true and deactivating itself when it has fired). The following example shows how a delay can be constructed that waits for a specific date to resume the task with which it is associated.

```
DELAY-FOR-CHANGING-DATES
  A DELAY
     EVENT (>> day of current date)

DELAY-UNTIL-DATE
  A DELAY-FOR-CHANGING-DATES
     DATE a date
     DELAY-UNTIL (>> (equal (>> date)) of current-date))
```

A *delay-for-changing-dates* checks its delay-until condition every time the day of current date changes. Its delay-until can be an arbitrary boolean condition. A *delay-until-date* has a more specific condition. It restarts its activity when it notices that the current-date is equal to the filler of its own subject *date*.

Delays can and have been used to define more complex and abstract tasks such as *monitor-deadline*, *wait-for-reply*, *wait-for-approval* etc.

4.3. Unification on Feature Sets

4.3.1. Feature Set Notations

KRS concepts describing office knowledge show a clear resemblance to a feature notation, which is one of the most conceptually clear formalisms. In a feature notation, objects and situations are described as sets of ordered pairs, where each pair consists of an attribute and a value. We will call such a set of pairs a *description*. Values of feaures can be atomic (i.e. a symbol) or complex (i.e. another feature). The realisation of feature formalisms in KRS is straightforward: attributes are KRS subject names, and values are subject-fillers. The expressive power of the notation can be enriched by allowing negation (represented by 'NOT') and disjunction (represented by curly braces). Notice that KRS definite descriptions can be used as values, as long as they evaluate to an atomic or a complex value. An example office concept in feature notation could be the following:

```
MY-TRIP
   TYPE TRIP
   REQUESTOR MILLIGAN
   DESTINATION PARIS
   DEPARTURE
        TYPE DATE
        DAY    22
        MONTH   6
        YEAR 1987
   DURATION
        DAYS {2 3}                ;;; Two OR three days
   TRAVEL-MEANS
        NOT CAR                   ;;; Any travel means but a car
   APPROVAL-AUTHORITY
       (>> MANAGER OF REQUESTOR)  ;;; A definite description
   REASON
        TYPE TECHNICAL-MEETING
        WITH SELLERS
        PROJECT P-82
```

Problem solving with feature descriptions can be done through *unification*, an operation on feature sets which is very successful in current (computational) linguistics, but which can be usefully applied to other domains as well. Other operations on feature descriptions can be envisioned (e.g. generalisation; Karttunen, 1986) but we will not explore their use in office modelling here.

The type of unification which we will describe here is related to Prolog-like unification (a kind of pattern matching in which variables in the patterns matched are bound such that they become equal), but with important differences. The type of unification we use was introduced into linguistics by Kay (1979, for an introduction, see Shieber, 1986). To our knowledge, this kind of unification has not yet been applied to office problem solving.

Unification is the union of feature sets, but with two important differences: there is a possibility of failure, and unification is structure-changing; if unification succeeds, the descriptions unified are changed in the process (they are merged). The resulting description is the smallest (most general) description subsuming all descriptions which have been unified. If unsuccessful, the operands of the unification operation are left unchanged. In unifying two feature sets, only the values of those attributes which are present in both feature sets are compared. If two values are atomic, they must be equal or the unification fails. If they are complex (i.e. feature sets), the unification operation is applied recursively on them. E.g., unifying the earlier description with

```
TRIP-INFO
    TYPE TRIP
    REASON
       PROJECT {IWONL NFWO}
    REPORT OBLIGATORY
```

would result in failure, as REASON PROJECT P-82 cannot be unified with REASON PROJECT {IWONL NFWO}. However, unification of the earlier description with

```
TRIP-INFO
    TYPE TRIP
    REASON
      PROJECT {P-82 P-440}
    REPORT OPTIONAL
```

would result in the merger of both descriptions, because the unification succeeds. In this case, the result would be the addition of the feature REPORT OPTIONAL to the initial description.

The main advantages of a feature notation combined with unification are declarativeness and order-independence. Declarative descriptions are simpler to understand and nearer to human thought than procedural descriptions. Order-independence means that the order in which descriptions are unified is irrelevant (unless side-effects are allowed). This implies that the amount of control information and execution sequence information needed is small. Furthermore, as there is a direct mapping from the feature notation to the KRS concept graph (or rather a part of it), the same advantages applying to the KRS concept graph (lazy evaluation, caching, consistency maintenance and inheritance) also apply to the feature notation. The fact

that descriptions can be *named*, and used with this name in other descriptions allows modularity and conciseness in the descriptions.

4.3.2. Applications of Unification

In the office problem-solving environment, unification can be used for the control of choices among alternatives, for structure building (specialising abstract descriptions to more concrete descriptions and completing partial descriptions) and inconsistency checking. We will explain these applications in turn.

(1) **Choice between alternatives.** Often, a choice must be made between different alternative tasks or activities. Tasks and activities may be extended with pre-conditions and post-conditions. Pre-conditions are checked against a description of the current situation. If they are true, the activity is applicable and can be executed. Post-conditions are conditions which should be true after the application of the activity. They can be used to check whether the activity was applied correctly or to prevent the execution of an activity the post-conditions of which are already satisfied. When at some point a choice must be made between two or more alternative activities or tasks, a correct choice can be made by unifying a description representing the current situation with the pre-conditions of each activity. If the unification fails, the activity cannot be executed, if it succeeds, execution is possible. Simultaneously with this unification, the description of the current situation is extended with information present in the pre-conditions of the activity. Analogously, unification of the current situation with the post-conditions of activities can be used to check whether the intended effect of an activity has been achieved.

(2) **Structure-building.** Initially vague, incomplete and abstract (general) descriptions can be gradually provided with more detail (specialised) by unifying them with descriptions representing the office knowledge. We might start, for example, with a *travel-request* by creating a concept which simply states our intention to travel, our destination and the time-period. The office problem solving mechanism may then provide such detail as the project the applicant works on, his manager, the preferred travel means, the amount of money to be given in advance and other information necessary for obtaining approval, finding financial support, contacting a travel agent etc., by unifying the initial and intermediate descriptions with descriptions embodying office knowledge.

(3) **Inconsistency checking.** Office work is distributed (different descriptions relating to the same task may be created by different people or at different times) and therefore prone to inconsistencies. Different descriptions can be checked for consistency by unifying them. A useful property of unification systems in this respect is that when a unification fails, the place of the inconsistency (however deeply recursively embedded) can be easily reported to the office worker.

5. CONCLUSION

We have shown that a flexible knowledge representation system and special-purpose formalisms developed for office procedure assistance are necessary prerequisites in the development of an intelligent workstation. The implementation of a prototype office environment in KRS was instrumental in the isolation of useful formalisms like priority-agendas, monitors and unification. Further experimentation with this prototype is expected to yield additional office formalisms.

6. REFERENCES

Ader, M. and M. Tueni. 'An Office Assistant Prototype.' ESPRIT Technical Week, Brussels, September 1987.

Fikes, R. and Kehler, T. 'The Role of Frame-Based Representation in Reasoning.' *Communications of the ACM*, Vol. 28 nr. 3 (1985)

Greiner, R. 'RLL-1: A Representation Language Language.' Stanford Heuristic Programming Project, HPP-80-9. Stanford University, California, 1980.

Haase, K. 'ARLO - Another Representation Language Offer' *MIT Bachelor's Thesis*, October 1986

Karttunen, L. 'Features and Values' *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford California, 1986.

Kay, M. 'Functional Grammar' in *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, Berkeley California, 1979.

Lieberman, H. 'Using Prototypical Objects to Implement Shared Bahavior in Object Oriented Systems.' in *Proceedings of OOPSLA'86'*, Portland Oregon, 1986.

Maes, P. *Computational Reflection*. Technical Report 87_2, A.I.-LAB, University of Brussels, 1987.

Shieber, S. *An Introduction to Unification-based Approaches to Grammar*. Chicago: University of Chicago Press.

Steels, L. 'The KRS concept system.' Technical Report 86_1, A.I.-LAB, University of Brussels, 1986.

Steels L., W. Van de Velde, J. Paredis, K. Van Marcke and V. Jonckers. 'Report on KRS Formalisms.' IWS deliverable D2/r2, A.I.-LAB, University of Brussels, 1987.

Tueni M., J. Z. Li and P. Fares. 'Supporting Execution and Monitoring of Office Tasks and Administrative Procedures.' IWS deliverable E/r1, Bull, 1987.

Van Marcke, K. 'KRS manual.' University of Brussels, A.I.-MEMO 87_3, 1987.

Van Marcke, K. 'Context Determination Through Inheritance in KRS.' University of Brussels, A.I.-MEMO 87_1, 1987.