

**Final thesis**

# **Extending the Knowledge Machine**

by  
**Markus Ingevall**

LITH-IDA-EX-05/026-SE

2005-03-14



**Final thesis**

# **Extending the Knowledge Machine**

by **Markus Ingevall**

LiTH-IDA-EX-05/026-SE

Supervisor: **Professor Erik Sandewall**  
IDA  
at Linköpings universitet

Examiner: **Professor Erik Sandewall**  
Department of Computer and Information  
Science  
at Linköpings universitet



## Abstract

This master's thesis deals with a frame-based knowledge representation language and system called The Knowledge Machine (KM), developed by Peter Clark and Bruce Porter at the University of Texas at Austin. The purpose of the thesis is to show a number of ways of changing and extending KM to handle larger classes of reasoning tasks associated with reasoning about actions and change.

**Keywords:** Knowledge representation, reasoning about actions and change, The Knowledge Machine, KM



## Acknowledgements

I would like to thank my supervisor and all members of the Cognitive Autonomous Systems Lab for their support during my work as well as Ola Leifler who has provided the L<sup>A</sup>T<sub>E</sub>X template used for producing this thesis.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Purpose and delimitations . . . . .	1
1.3	Reading guidelines . . . . .	2
<b>2</b>	<b>Description of KM</b>	<b>3</b>
2.1	Basics of KM . . . . .	3
2.2	Situation Mechanism . . . . .	7
2.3	Querying the knowledge base . . . . .	13
2.4	Structure of the KM System . . . . .	19
2.5	Use of KM so far . . . . .	20
<b>3</b>	<b>Limitations and extensions</b>	<b>21</b>
3.1	Limitations of KM . . . . .	21
3.2	Extensions . . . . .	24
<b>4</b>	<b>Postdiction</b>	<b>29</b>
4.1	Inspiration . . . . .	29
4.2	Implementation . . . . .	29
4.3	Usage . . . . .	31
<b>5</b>	<b>Checking effects and preconditions</b>	<b>33</b>
5.1	Inspiration and implementation . . . . .	33
5.2	Usage . . . . .	34

---

<b>6</b>	<b>Changes to action definitions affecting instance-slot-value relations</b>	<b>37</b>
6.1	Inspiration . . . . .	37
6.2	Implementation . . . . .	38
6.3	Usage . . . . .	38
<b>7</b>	<b>Non-deterministic direct effects</b>	<b>41</b>
7.1	Inspiration . . . . .	41
7.2	Implementation . . . . .	41
7.3	Usage . . . . .	43
<b>8</b>	<b>Concurrent actions</b>	<b>45</b>
8.1	Inspiration . . . . .	45
8.2	Implementation . . . . .	45
8.3	Usage . . . . .	47
<b>9</b>	<b>New way of modelling actions</b>	<b>51</b>
9.1	Inspiration . . . . .	51
9.2	Implementation . . . . .	52
9.3	Usage . . . . .	55
<b>10</b>	<b>Failed actions and sequences of actions</b>	<b>59</b>
10.1	Inspiration . . . . .	59
10.2	Implementation . . . . .	59
10.3	Usage . . . . .	60
<b>11</b>	<b>Situation levels</b>	<b>69</b>
11.1	Inspiration and implementation . . . . .	69
11.2	Usage . . . . .	71
<b>12</b>	<b>Results and future work</b>	<b>73</b>
<b>A</b>	<b>Dictionary</b>	<b>75</b>
<b>B</b>	<b>New concepts in KM</b>	<b>79</b>

# Chapter 1

## Introduction

### 1.1 Background

The inspiration for this master's thesis has largely been two courses in artificial intelligence given at Linköping Institute of Technology in 2003 and 2004. While the latter of the two specifically treated the area of interest of this thesis, knowledge representation, the first course gave an overview of artificial intelligence and the programming language Lisp. Knowledge of Lisp has later been found to be indispensable for performing the work on the thesis.

### 1.2 Purpose and delimitations

The purpose of this master's thesis is to change and extend a knowledge representation system called The Knowledge Machine (KM) to handle larger classes of reasoning tasks associated with reasoning about actions and change.

The purpose is not to change the basic structure of the KM system described in section 2.4 or to make a completely new implementation of the KM language. Thus, the limitations of the system discussed in the user manual [CPC] have not been considered. Instead, only changes and

extensions to the system related to the situation mechanism part of KM have been implemented.

Additionally, a starting point for work on this thesis has been to only take into consideration single world histories. Therefore, simulation of multiple, possible futures as described in the situations manual [CPb] is excluded.

### **1.3 Reading guidelines**

The intended reader of this master's thesis is a person interested either in knowledge representation in general or in working with or making changes to KM. Chapter 2 describes the original version of KM, including its use so far. Limitations and extensions are then discussed in chapter 3. The thesis continues by covering the implemented extensions in the subsequent chapters and concludes by summarising the results and discussing future work.

## Chapter 2

# Description of KM

### 2.1 Basics of KM

KM, The Knowledge Machine, is primarily the name of a frame-based knowledge representation language developed by Peter Clark and Bruce Porter at the University of Texas at Austin [CPc]. However, KM is also used to denote the Lisp implementation of the language.

The basic unit of representation in KM is a *frame*, which may contain *slots*. Each slot is associated with *values*, which may be atomic, or expressions referring to the same and/or other frames. The storage of *frame-slot-values* relations in the KM system is denoted the knowledge base. There are two kinds of frames, *instances* (individuals) and *classes* (types of individuals). In order to describe the properties of the members of a class, expressions of the following form are used:

```
(every <class> has
      (<slot1> (<expr11> <expr12> ...))
      (<slot2> (<expr21> <expr22> ...))
      ...)
```

The slots ( $\text{slot}_i$ ) may be regarded as binary relations, which hold between the instances of the class and other instances. The expressions ( $\text{expr}_{ij}$ )

evaluate to zero or more instances. Even though KM statements can usually be translated into standard first-order logic notation, KM does not use such a notation, as the above example clearly shows. By not using FOL notation, the authors of KM have intended to make knowledge base construction easier. However, as stated most KM expressions may be translated into FOL notation. In the above case, the equivalent would be

$$\forall x, y \text{ isa}(x, \text{class}) \wedge y \in \text{expr}_{ij}(x) \rightarrow \text{slot}_i(x, y)$$

where  $\text{expr}_{ij}(x)$  is a function which returns the value of  $\langle \text{expr}_{ij} \rangle$  for some instance  $x$  of  $\text{class}$  and  $\text{isa}(x, c)$  is true if  $x$  is a member of class  $c$ .

The properties of a particular instance is described in a similar manner:

```
(<instance> has
  (instance-of (<class1> ... <classn>))
  (<slot1> (<expr11> <expr12> ...))
  (<slot2> (<expr21> <expr22> ...))
  ...)
```

In this case, the slot relations hold between the instance and the values acquired by evaluating the expressions. The values of the `instance-of` slot are the names of the classes to which the instance belongs. The FOL equivalent is

$$\forall I, y \ y \in \text{expr}_{ij}(I) \rightarrow \text{slot}_i(I, y)$$

Here,  $\text{expr}_{ij}(I)$  returns the value of  $\langle \text{expr}_{ij} \rangle$  for the instance  $I$ .

Classes can have properties of their own, which are not inherited by their members. Most importantly, classes may have superclasses and subclasses, representing supersets and subsets of members of the class. Class properties are declared in the following way:

```
(<class> has
  (superclasses (<superclass1> ... <superclassn>))
  (<slot1> (<expr11> <expr12> ...))
  (<slot2> (<expr21> <expr22> ...))
  ...)
```

The slots `superclasses` and `subclasses` are inverses of each other and declaration is only needed one way. If, for instance, the class `Car` is declared to be the subclass of `Vehicle` then `Vehicle` will be the superclass of `Car`.

Thus, a class for bananas could be described in the following way:

```
(Banana has
  (superclasses (Fruit)))
```

```
(every Banana has
  (colour (*Yellow))
  (shape (*Bent)))
```

The FOL equivalents would be

$$\begin{aligned} & \text{superclasses}(\textit{Banana}, \textit{Fruit}) \\ \forall b \textit{ isa}(b, \textit{Banana}) & \rightarrow \textit{colour}(b, *Yellow) \wedge \textit{shape}(b, *Bent) \end{aligned}$$

The `*` prefix in `*Yellow` and `*Bent` above is an example of the KM naming conventions:

<b>classes</b>	begin with an upper-case letter, eg. <code>Banana</code>
<b>slots</b>	begin with a lower-case letter, eg. <code>colour</code>
<b>named instances</b>	begin with a <code>*</code> prefix, eg. <code>*Yellow</code>
<b>anonymous instances</b>	begin with a <code>_</code> prefix, eg. <code>_Car13</code>
<b>variables</b>	begin with a <code>?</code> prefix, eg. <code>?x</code>

As the naming conventions propose there are two kinds of instances in KM, named and anonymous ones. The latter are usually created by the system at run-time while evaluating expressions whose FOL equivalents would include existential quantifiers ( $\exists$ ). In contrast to named instances, the anonymous ones may be unified with other instances. When instances are unified in KM they are asserted to be equal, i.e. they are asserted to refer to the same instance. Obviously, in order for unification to work the slot values of the instances being unified must not conflict, i.e. must be equal or unifiable, and, as mentioned, named instances may not be unified with each other as they are regarded to be unique.

To create a named instance of the `Banana` class defined earlier one might write

```
(*The-banana has
      (instance-of (Banana)))
```

Slots are represented as instances of the built-in class `Slot` and are associated with a number of built-in slots which specify their properties. There are six basic ones:

**domain** specifies the most general class(es) allowed for an instance using the slot.

**range** specifies the most general class(es) allowed for the filler of the slot.

**cardinality** can take one of four values: `1-to-1`, `1-to-N`, `N-to-1` or `N-to-N`. Their meaning is most easily explained by giving examples of slots having the different kinds of cardinality, namely `spouse`, `is-father-of`, `has-father` and `has-friends`. A person can be married to at most one other person. A father can have many children but each child only has one father. A person can have many friends and they can have many friends of their own. The default cardinality is `N-to-N`. `1-to-1` and `N-to-1` slots are denoted single-valued, whilst the other two kinds are called multivalued.

**inverse** For instance, `is-father-of` and `has-father` are each other's inverses. The default inverse of *slot* is *slot-of*.

**subslots** specifies more specific slots. For instance, `sweaters` might be a subslot of `clothes`.

**superslots** is the inverse of **subslots**, i.e. more general slots.

Slots do not have to be explicitly declared. If no declaration is provided for a certain slot its domain and range won't be restricted and it will acquire the default cardinality and inverse.

There are two main commands used for retrieving information from the knowledge base, (`showme <instance>`) and (`the <slot> of <instance>`). For the time being, it is sufficient to say that the `showme` command simply displays what facts about the specified instance are currently stored in the knowledge base whilst the other one computes the value of the specified slot in the given instance. In what way this is done is further explained in section 2.3. In the banana example given earlier, the command



---

(showme \*The-banana)

would produce the answer

(\*The-banana has  
                  (instance-of (Banana)))

whilst

(the colour of \*The-banana)

would return

(\*Yellow)

since \*The-banana is a banana and all bananas are yellow.

## 2.2 Situation Mechanism

Apart from its basic functionality, KM provides a situation mechanism based on, but not identical to, Situation Calculus [CPb]. (An introduction to Situation Calculus may be found in [RN03].) Its main concept is a *situation*, describing the state of the world at a certain moment. The "normal" knowledge base, which facts are entered into by default, is regarded as the global situation, and its facts are common to all other situations, i.e. if something holds in the global situation it holds in all other situations as well. In KM terms, the global situation is the default supersituation. However, other situations may be supersituations as well, in which case what is true in a supersituation also holds in all its underlying situations, its subsituations. Facts may also be situation-specific, which means that they hold in a certain situation but not necessarily in any other.

Situations are instances of the built-in class `Situation` and may be ordered in time by being connected to each other. A situation can have several next situations but only one previous. This reflects the notion that two or more situations cannot be followed by a single one, whilst multiple, possible futures are allowed. Next and previous situations are specified using the built-in slots `next-situation` and `prev-situation` respectively.

In order for some slots to be situation-dependent and others to be situation-independent they need to be grouped into different categories. This is done using the built-in slot `fluent-status`, which can have one of three values, `*Non-Fluent`, `*Fluent` or `*Inertial-Fluent`. A `*Non-Fluent` slot carries the same value in all situations. A `*Fluent` slot has different values depending on the situation. Lastly, the value of an `*Inertial-Fluent` slot may vary between situations but it persists from one situation to the next if no change affecting the value takes place. Note that using fluent status is KM's way of treating the frame problem, i.e. how to efficiently represent the fact that, in general, most things don't change but stay the same from one situation to another.

The state of the world may change either due to explicit slot value manipulation or as a result of *actions*. An action is an instance of the built-in class `Action` and takes place between two situations. The latter situation is created when the action is executed. The action is regarded as being instantaneous and thus there is no situation corresponding to the execution of the action. Actions are modelled using four lists represented by slots, `pcs-list` (preconditions list), `ncs-list` (negated preconditions list), `add-list` (add list) and `del-list` (delete list). `pcs-list` specifies what must hold for the action to take place and `ncs-list` specifies what must not hold. `add-list` contains propositions which become true as a result of the action and `del-list` contains propositions which become false. In FOL notation, this can be expressed in the following way:

$$\begin{aligned} \forall s, s', p, a \text{ holds-in}(\text{pcs-list}(a, p), s) \wedge \text{next-situation}(s, s', a) &\rightarrow \text{holds-in}(p, s) \\ \forall s, s', p, a \text{ holds-in}(\text{ncs-list}(a, p), s) \wedge \text{next-situation}(s, s', a) &\rightarrow \text{holds-in}(\neg p, s) \\ \forall s, s', p, a \text{ holds-in}(\text{add-list}(a, p), s) \wedge \text{next-situation}(s, s', a) &\rightarrow \text{holds-in}(p, s') \\ \forall s, s', p, a \text{ holds-in}(\text{del-list}(a, p), s) \wedge \text{next-situation}(s, s', a) &\rightarrow \text{holds-in}(\neg p, s') \end{aligned}$$

$\text{holds-in}(\text{prop}, s)$  denotes that  $\text{prop}$  holds in situation  $s$  and  $\text{next-situation}(s, s', a)$  means that  $s'$  is the next situation of  $s$  and that the situations are connected by action  $a$ . The above formulas state that the propositions of the preconditions list are *made* to hold and the propositions of the negated preconditions list are made not to hold. This corresponds to performing an action using the KM command `(do <action>)`. However, there is a possibility, using the `(try-do <action>)` command instead, of performing an action only if the propositions of the preconditions list hold and the

propositions of the negated preconditions list do not. In this case, the FOL notation changes into

$$\begin{aligned} \forall s, s', p_1, p_2, p_3, a \text{ next-situation}(s, s', a) \wedge \text{holds-in}(\text{add-list}(a, p_1), s) \wedge \\ \text{holds-in}(\text{pcs-list}(a, p_2), s) \wedge \text{holds-in}(\text{ncs-list}(a, p_3), s) \wedge \\ \text{holds-in}(p_2, s) \wedge \text{holds-in}(\neg p_3, s) \\ \rightarrow \text{holds-in}(p_1, s') \\ \forall s, s', p_1, p_2, p_3, a \text{ next-situation}(s, s', a) \wedge \text{holds-in}(\text{del-list}(a, p_1), s) \wedge \\ \text{holds-in}(\text{pcs-list}(a, p_2), s) \wedge \text{holds-in}(\text{ncs-list}(a, p_3), s) \wedge \\ \text{holds-in}(p_2, s) \wedge \text{holds-in}(\neg p_3, s) \\ \rightarrow \text{holds-in}(\neg p_1, s') \end{aligned}$$

The `do-and-next` and `try-do-and-next` commands work the same way as `do` and `try-do` respectively, except that the system stays in the new situation after performing the action.

Actions may be regarded to have indirect as well as direct effects, of which only the latter are stated in the action descriptions. The indirect effects, or ramifications, are determined in KM by conditions, or rules, associated with the affected slots. For instance, suppose there is an action type `Switch-on`. Its direct effect might be that a certain switch changes position, whilst as an indirect effect is that an electric current appears in a certain wire. Indirect effects may give rise to other indirect effects as well. For instance, the electric current might make a certain light go on. As this example aims to show, it is difficult, if not impossible, to know all the effects of an action if rules are used. One problem of using rules, and thus allowing ramifications, is that actions might have contradictory effects. Section 3.1 explains this further.

Below is an example with the aim of illustrating in what way the situation mechanism works. First, the following class and instance declarations are entered:

```
(every Vehicle has
      (has-wheels (*True)))
```

```
(Car has
      (superclasses (Vehicle)))
```

```
(*The-car has  
      (instance-of (Car)))
```

```
(*True has  
      (instance-of (Truth-value)))
```

```
(*Red has  
      (instance-of (Colour)))
```

```
(*Blue has  
      (instance-of (Colour)))
```

```
(*Yellow has  
      (instance-of (Colour)))
```

```
(*Green has  
      (instance-of (Colour)))
```

```
(*Purple has  
      (instance-of (Colour)))
```

```
(*Orange has  
      (instance-of (Colour)))
```

These are followed by the slot declarations:

```
(has-wheels has  
      (instance-of (Slot))  
      (fluent-status (*Non-Fluent)))
```

```
(speed has  
      (instance-of (Slot))  
      (fluent-status (*Fluent)))
```

```
(colours has  
      (instance-of (Slot))  
      (fluent-status (*Inertial-Fluent)))
```

```
(action-colours has
  (instance-of (Slot))
  (fluent-status (*Non-Fluent)))
```

```
(object has
  (instance-of (Slot))
  (fluent-status (*Non-Fluent)))
```

Next, three kinds of actions are specified:

```
(Remove-colour has
  (superclasses (Action)))
```

```
(every Remove-colour has
  (object ((a Car)))
  (action-colours ((a Colour)))
  (pcs-list (:(triple
    (the object of Self)
    colours
    (the action-colours of Self))))
  (del-list (:(triple
    (the object of Self)
    colours
    (the action-colours of Self)))))
```

```
(Paint has
  (superclasses (Action)))
```

```
(every Paint has
  (object ((a Car)))
  (action-colours ((a Colour)))
  (add-list (:(triple
    (the object of Self)
    colours
    (the action-colours of Self)))))
```

```
(Repaint has
  (superclasses (Action)))
```

```
(every Repaint has
  (object ((a Car)))
  (action-colours ((a Colour)))
  (del-list ( (:triple
               (the object of Self)
               colours
               (the colours of (the object of Self))))))
  (add-list ( (:triple
               (the object of Self)
               colours
               (the action-colours of Self))))))
```

In these action definitions the expressions (a Car) and (a Colour) are used. An expression of this kind creates an anonymous instance of the specified class, an instance, which may later be unified with another instance. `Self`, used in for instance (the object of Self), refers to the object itself, in this case the action instance. The `:triple` expressions are primarily used in action definitions but can be used separately as well. (`:triple <instance> <slot> <value>`) represents the fact that the slot of the instance carries the specified value. However, it is not an assertion of the fact. To enter the fact into the knowledge base one would write

```
(assert (:triple <instance> <slot> <value>))
```

This is equivalent to

```
(<instance> has
  (<slot> (<value>)))
```

To enter "situation mode" the command (`new-situation`) is passed to KM. This creates an anonymous instance of `Situation` and the system enters it, i.e. it is regarded as the current situation. Asserting facts in the new situation is done in the same way as in the global situation. One might for instance write

```
(*The-car has
  (colours (*Red *Blue))
  (speed (70)))
```

To perform each of the actions of the example once one could write

```
(do-and-next (a Paint with
              (object (*The-car))
              (action-colours (*Green))))
```

```
(try-do-and-next (a Remove-colour with
                  (object (*The-car))
                  (action-colours (*Red))))
```

```
(do-and-next (a Repaint with
              (object (*The-car))
              (action-colours (*Yellow *Purple))))
```

The `try-do-and-next` command is used only in the case of (`Remove-colour`). This is because the other action types lack preconditions. One could use `try-do-and-next` to perform them as well but it would never make any difference.

The first of the following three commands adds the value 90 to the `speed` slot of `*The-car`. The second one creates a new situation without performing any action, connects to the current one and enters it. The third command adds the values `*Blue` and `*Orange` to the `colours` slot.

```
(*The-car has
  (speed (90)))
```

```
(next-situation)
```

```
(*The-car has
  (colours (*Blue *Orange)))
```

Section 2.3 uses the above example to explain in what way the system computes slot values.

## 2.3 Querying the knowledge base

As mentioned in section 2.1, the command (`the <slot> of <instance>`) returns the value of the slot in the specified instance [CPC]. In KM, the

function used for handling commands is called `km0` [CPa]. It handles commands by matching them to a list containing command patterns and "instructions" on how to handle commands of each kind. In the case of slot value computation, the function `km-slotvals-from-kb` is called. It uses a number of sources to compute the value:

- Prototypes
- Projection
- Subslots
- Local values (values from the current situation and its supersituations)
- Class inheritance

The function first adds in values from prototypes. However, this thesis does not concern itself with prototypes and they will therefore not be explained further. More information can be obtained in [CPc].

The local values are obtained by merging values from the current situation and its supersituations in the cases of `*Fluent` and `*Inertial-Fluent` slots. If the slot is `*Non-Fluent` values are fetched only from the global situation. The resulting values are unified together, if possible, and filtered through any constraints found. If the slot is single-valued and the unification results in more than one value no consistent value can be found and the system therefore produces an error message and deletes all values of the slot in the current situation.

If the fluent status of the slot is `*Inertial-Fluent` and the query is not passed from the global situation the system tries to project values from previous situations, or rather makes a `km0` request for the value of the slot in the previous situation. All previous situations are thus investigated by means of recursion. The values found are filtered through any local constraints.

For each of the slot's subslots, values are obtained by calling `km0`. The subslot values, as well as projected values if the slot is multivalued, are then unified with the local ones as far as possible. Subslot values, which cannot be unified with local values are simply added if the slot is multivalued.



If it is single-valued and the unification produces more than one value no consistent value can be found and, as in the case of the local values, the system produces an error message and deletes all values of the slot in the current situation.

At this point, the values found so far are saved in the knowledge base. Then values are fetched from the instance's classes and unified in. As before, if the slot is single-valued the system produces an error message and deletes all values of the slot in the current situation. The reason for saving the values before getting the class values is that not doing so would sometimes result in indefinite looping.

As mentioned before, the projected values are unified with the local and subslot values only if the slot is multivalued. If it is single-valued projection will result in a single value provided it is successful. The system tries to unify that single value with what has already been found. If it does not succeed the projected value is discarded. Lastly, for both kinds of slots the produced value or values are saved in the knowledge base.

The slot-value computation may be expressed in a FOL-like manner. In the basic case where the situation mechanism is not used, i.e. where all slots may be regarded as *\*Non-Fluent* it can be written in the following way:

$$\begin{aligned} ANS &= \{x \mid slot(instance, x)\} \\ &= unify-if-sv(OWN-VALUES, slot) \cup_{KM} \\ &\quad unify-if-sv(SUBSLOT-VALUES, slot) \cup_{KM} \\ &\quad unify-if-sv(INHERITED-VALUES, slot) \end{aligned}$$

where

$$OWN-VALUES = \{x \mid own-value(instance, slot, x)\}$$

$$SUBSLOT-VALUES = \{x \mid subslot(slot, sub) \wedge sub(instance, x)\}$$

$$INHERITED-VALUES = \{x \mid isa(instance, c) \wedge member-value(c, slot, x)\}$$

*own-value(instance, slot, value)* is true if and only if the knowledge base contains the relation *instance-slot-value*. *subslot(slot, sub)* means that *sub* is a subslot of *slot* and *member-value(class, slot, value)* holds if and only if the *slot-value* relation holds for all members of *class*. *own-value*,

*subslot(slot, sub)*, and *member-value* are not defined in the KM manual but are used here to make it easier to explain the slot-value computation procedure. The same holds for *unify-if-sv(set, slot)*, which attempts to unify the members of the given set provided that the slot is single-valued. *ANS* is used to denote the result of the query.  $\cup_{KM}$  is a means of expressing KM's unification of sets. When sets are unified each set member is unified with at most one member of another set. Members of the same set are not unified. The resulting set will thus contain at least the same number of members as the smallest of the unified sets. When two instances are unified during set unification, just as in the case of separate unification of instances, both of them must not be named instances and their slot values must be equal or unifiable. However, for two instances to be unified during set unification an additional condition must hold. The classes of one of the instances must subsume or equal the classes of the other instance, i.e. the instances must be of the same "kind" or one of them has to be of a more general kind than the other. If unification fails the KM system will report an error and no result will be given.

In order for the situation mechanism to be included, some changes and additions are required. The computation of slot values may now be expressed in the following way:

$$\begin{aligned} VALUES &= \{x | holds-in(slot(instance, x), sit)\} \\ &= unify-if-sv(LOCAL-VALUES, slot) \cup_{KM} \\ &\quad unify-if-sv(SUBSLOT-VALUES, slot) \cup_{KM} \\ &\quad unify-if-sv(INHERITED-VALUES, slot) \cup_{KM} \\ &\quad \{y | y \in PROJECTED-VALUES \wedge \neg single-valued(slot)\} \end{aligned}$$

where

$$\begin{aligned} LOCAL-VALUES &= \{x | [fluent-status(slot, *Non-Fluent) \wedge \\ &\quad own-value(instance, slot, x)] \vee \\ &\quad [\neg fluent-status(slot, *Non-Fluent) \wedge \\ &\quad own-or-super-situation-value(instance, slot, x, sit)]\} \end{aligned}$$

$$SUBSLOT-VALUES = \{x | subslot(slot, sub) \wedge holds-in(sub(instance, x), sit)\}$$

$$INHERITED-VALUES = \{x | isa(instance, c) \wedge member-value(c, slot, x)\}$$

---

$PROJECTED-VALUES =$   
 $unify-if-sv(\{x \mid fluent-status(slot, *Inertial-Fluent) \wedge$   
 $sit \neq *Global \wedge$   
 $prev-situation(sit, sit') \wedge$   
 $holds-in(slot(instance, x), sit')\})$

and

$[\neg single-valued(slot) \rightarrow ANS = VALUES] \wedge$   
 $[single-valued(slot) \wedge$   
 $unifiable(VALUES, PROJECTED-VALUES)$   
 $\rightarrow ANS = VALUES \cup_{KM} PROJECTED-VALUES] \wedge$   
 $[single-valued(slot) \wedge$   
 $\neg unifiable(VALUES, PROJECTED-VALUES)$   
 $\rightarrow ANS = VALUES]$

$fluent-status(slot, fs)$  is true if and only if the fluent status of the slot is  $fs$ .  
 $own-or-supersituation-value(instance, slot, value, sit)$  holds if and only if  
the knowledge base contains the relation  $instance-slot-value$  for situation  
 $sit$  or any of its supersituations. It should be noted that, when using the  
situation mechanism,  $own-value$  defined earlier is only applicable to **\*Non-  
Fluent** slots, which carry the same values in all situations. If the situation  
mechanism is not used the fluent status does not affect the computation of  
slot values.

As mentioned earlier,  $holds-in(prop, sit)$  is used to denote that  $prop$   
holds in the situation  $sit$ . For instance, if a query for the values of the slot  
`colours` of the instance `*The-car` in the situation `_Situation54` produces  
the result `(*Red *Blue)` then  $holds-in(slot(*The-car,*Red),\_Situation54)$   
and  $holds-in(slot(*The-car,*Blue),\_Situation54)$  are true.

$prev-situation(sit, sit')$  is true if and only if  $sit'$  is the previous situation  
of  $sit$  and  $unifiable$ , which can take an arbitrary number of sets, is true  
if and only if the sets can be unified. In the example from section 2.2, the  
command `(showme *The-car)` produces the following result:

```

(*The-car has
  (instance-of (Car))
  (has-wheels (*True))

```

```
(object-of (_Paint2
            _Remove-colour4
            _Repaint6)))

(in-situation _Situation1
  (*The-car has
    (colours (*Red
              *Blue))
    (speed (70))))

(in-situation _Situation3
  (*The-car has
    (colours (*Red
              *Blue
              *Green))))

(in-situation _Situation5
  (*The-car has
    (colours (*Blue
              *Green
              (<> *Red)))))

(in-situation _Situation7
  (*The-car has
    (colours (*Yellow
              *Purple
              (<> *Blue)
              (<> *Green)))
    (speed (90))))

(in-situation _Situation8
  (*The-car has
    (colours (*Blue
              *Orange))))
```

The command (the has-wheels of \*The-car) will give the same result, (\*True), regardless of what situation the system is currently in since has-wheels is a \*Non-Fluent slot. However, that is not the case for colours

and `speed` whose values vary between situations. Since `speed` is a `*Fluent` slot the system won't fetch values for it in any other situations. For instance, in `_Situation7` the command (the `speed` of `*The-car`) will return the value 70, whilst it will give NIL, i.e. no value, in `_Situation8`.

In the example, `_Paint2`, an anonymous instance of `Paint`, is executed in `_Situation1` producing `_Situation3` and adding `*Green` to the `colours` slot. It is followed by `_Remove-colour4` leading to `_Situation5` and adding (`<> *Red`) to the `colours` slot, i.e. prohibiting the instance-slot combination to take the value `*Red` in `_Situation5`. The last action to be performed is `_Repaint6`, which removes the present colours `*Blue` and `*Green` and adds the specified `*Yellow` and `*Purple`. In the last situation, the colours `*Blue` and `*Orange` are added. Considering that `colours` is an `*Inertial-Fluent` slot, one might wonder why it does not carry the values `*Yellow` and `*Purple` from `_Situation7` in `_Situation8`. The reason is that the underlying principle in KM is never to compute anything unless computation is explicitly requested. Thus, `*Yellow` and `*Purple` are not automatically projected from `_Situation7` to `_Situation8`. However, if a request for the value is passed in `_Situation8` by giving the command (the `colours` of `*The-car`) it will result in the answer (`*Yellow *Purple *Blue *Orange`).

Slot value requests may be more complex than the ones presented above. Suppose, for instance, that `*Blue` carries the slot `nuance` with the value `*Sky`. Then, in `_Situation8`, the command (the `nuance` of (the `colours` of `*The-car`)) will give the answer `*Sky`. Slot value commands may be even more complex, which is described in [CPc] and [CPb].

## 2.4 Structure of the KM System

The KM system can be divided into three main parts: command processing, computation, and knowledge base handling [CPa]. This division is not explicitly described in the code or the manuals but appears apparent when the code is examined. As mentioned, commands in KM are matched to a list containing command patterns and instructions on how to handle the different kinds of commands. The function responsible for performing the matching is called `km0`.

In terms of amount of code, the computation part is the largest of the three and the main area of interest for this thesis. As the name suggests, it is where the "reasoning" in KM is performed, for instance slot value computation. The last of the three parts is responsible for maintaining and handling the KM knowledge base, i.e. what has been entered into the system. Apart from these three parts, there are some auxiliary functions used for loading and saving knowledge base files and resetting the knowledge base and two others used for displaying the version number and licence.

## 2.5 Use of KM so far

KM has mainly been used in three projects. The first of these was the Botany Knowledge Base project, which was conducted from 1986 to 1994 and which generated the first early version of KM. According to the project home page [bkb05], the goal was "to create a laboratory for research on AI tasks that require extensive knowledge, such as novel problem solving, language understanding, and learning".

Between 1999 and 2003, KM was used by the SRI team participating in the Rapid Knowledge Formation (RKF) project. The aim of the project was to enable experts of different fields untrained in AI to construct knowledge bases efficiently and accurately. The SRI team, made up of Boeing, Information Sciences Institute (ISI) at University of Southern California, Northwestern University, Stanford University, University of Massachusetts at Amherst, University of Texas at Austin, University of West Florida, Massachusetts Institute of Technology, and Pragati Systems, developed the SHAKEN system. The group home page [sri05] is still maintained, in contrast to the project home page. As part of the project, the possibility of creating and executing rather elaborate plans was introduced. It is not covered in the manuals but is explained in chapter 10. Currently, KM is being used in the Halo project [hal05] whose goal is similar to that of the RKF project.

Since these projects have not been mainly concerned with reasoning about action and change they have not been a major source of inspiration for this master's thesis. In fact, nothing has been found indicating that extensions similar to those of this thesis have been made to KM before.

## Chapter 3

# Limitations and extensions

### 3.1 Limitations of KM

The KM manual [CPc] describes some known limitations not having to do with the situation mechanism. As mentioned in section 1.2 these are not within the area of interest of this thesis. Instead, this thesis concerns itself with KM's situation mechanism. The situations manual [CPb] contains a discussion on known limitations of the situation mechanism, which include

- Chronological minimization
- Lack of projection back in time
- Disjunctive ramifications
- Modelling of continuous change
- The situation-action dichotomy

As explained in section 2.2, for **\*Inertial-Fluent** slots KM will search for values in all previous situations, the reason for this being that if an instance-slot-value relation holds in a certain situation then it may be presumed to

hold in later situations as well provided that it is not contradicted. Now suppose that in a certain situation it is known that a cup is on a table, that in a later situation it is no longer there and that no action has been performed to remove the cup from the table. In this case, KM will draw the conclusion that the cup is on the table in all situations up to the one preceding the situation where the cup is known not to be on the table. This is obviously too strong a conclusion to draw. In other words, KM assumes changes to take place as late as possible, performing so-called chronological minimization.

If an instance-slot-value relation is known to hold in a future situation and no action is known to have been performed making the relation hold then it could be assumed that the relation holds in earlier situations as well. However, KM is not capable of drawing such conclusions.

As mentioned in section 2.2, if rules are used in KM, and thus ramifications are allowed, this may give rise to contradictory effects of actions. For instance, suppose there is a person with dual citizenship of country A and B who has to choose one citizenship at the age of 18. Thus, if she is a citizen of country A at the age of 18 then she will no longer be a citizen of country B and vice versa. In such a case, what is her citizenship at the age of 18? In KM, the answer will depend on the modelling and in what order queries are passed. The scenario could be set up in the following way:

```
(age has
  (instance-of (Slot))
  (fluent-status (*Fluent))
  (cardinality (N-to-1)))

(citizen-of-A has
  (instance-of (Slot))
  (fluent-status (*Inertial-Fluent))
  (cardinality (N-to-1)))

(citizen-of-B has
  (instance-of (Slot))
  (fluent-status (*Inertial-Fluent))
  (cardinality (N-to-1)))
```



```
(*The-person has
  (citizen-of-A ((if ( ((the age of Self) > 17)
                      and ((the citizen-of-B of Self) = *Yes))
                    then *No
                    else *Yes)))
  (citizen-of-B ((if ( ((the age of Self) > 17)
                      and ((the citizen-of-A of Self) = *Yes))
                    then *No
                    else *Yes))))
```

```
(new-situation)
```

```
(*The-person has
  (age (17))
  (citizen-of-A (*Yes))
  (citizen-of-B (*Yes)))
```

```
(next-situation)
```

```
(*The-person has (age (18)))
```

Here, when queries are passed for `citizen-of-A` and `citizen-of-B` KM will return `*No` as a response to the first query and `*Yes` as a response to the second one. It should be noted that the direct effects of actions are explicitly not allowed to be disjunctive. It is thus not possible to model non-deterministic actions.

Another limitation of KM is that it does not offer any means of modelling continuous change other than by representing it in the form of several discrete actions. Such a representation is not sufficient in all cases. Furthermore, KM makes a clear distinction between actions and situations, not allowing modelling of things happening during the execution of an action, i.e. actions are presumed to be instantaneous. In addition to this, execution of actions is limited to one at a time. Thus, concurrent actions are not allowed. Additionally, there is no efficient way of modelling delayed effects of actions.

In KM, even though an action's definition may be changed after it has been performed the changes won't affect any instance-slot-value relations.

If the action is executed again, using for instance the (do <action>) command, a new situation is created in which the effects are asserted, leaving the old next situation unaffected.

Since instance-slot-value relations may be added or removed at any time in the global or any other situation instance-slot-value relations may sometimes contradict the effects and preconditions of the actions which have been performed. However, KM is unable to detect such contradictions and will take no notice of the effects and preconditions once the actions have been performed.

## 3.2 Extensions

A number of possible ways of extending KM have been considered, partly inspired by the limitations discussed in section 3.1:

- Projection back in time (postdiction)
- Check of effects and preconditions
- Changes to action definitions affecting instance-slot-value relations
- Non-deterministic direct effects
- Concurrent actions
- New way of modelling actions
- Failed actions and sequences of actions
- Situation levels
- Continuous change

Each of these is discussed briefly below and those implemented are further described in the following chapters.

As described in section 2.2, KM allows a situation to be followed by more than one other situation, thus providing the possibility of modelling multiple, possible futures. However, such modelling is outside the area of interest of this master's thesis and thus, each situation is allowed to have at

most one next situation. Note that this does not affect the FOL formulas associated with performing actions presented in section 2.2.

Projection back in time, also known as postdiction, has been implemented and is covered in chapter 4. In short, values are fetched from a future situation if the preceding action does not affect them. If projected and postdicted values cannot be combined with values from other sources the values from the other sources are used and an error message is passed.

As described in section 3.1, contradictions may appear between instance-slot-value relations on the one hand and the effects and preconditions of actions on the other. In the new version resulting from this thesis this is handled in connection to querying by passing error messages when contradictions are discovered.

The possibility of letting changes to action definitions affect instance-slot-value relations has been implemented by changing the way actions are performed. Simply put, if the situation in which a certain action is performed already has a next situation no new situation is created. Instead, the effects of the action are made to hold in the existing next situation.

The situations manual [CPb] stating that the original version of KM explicitly prohibits the direct effects of actions to be disjoint, i.e. non-deterministic, has inspired looking into the question of introducing non-deterministic direct effects. As mentioned in section 3.1, non-determinism may occur in the original version of KM due to disjunctive ramifications. Handling this kind of non-determinism would require extensive computations considerably slowing down the system. Therefore, no effort has been put into it. However, it is imaginable that there may be a need to represent actions whose effects are partly but not completely known. Suppose for instance that it is known that a six-sided die is tossed but that the result of the toss is unknown. Even though one cannot say what number turned up it is known that it was either one, two, three, four, five, or six. Thus, one of these values should be present in the next situation. If it is not an error message is passed.

The extensions related to concurrent actions and a new way of modelling actions have both been inspired by [Gus01]. In the original version of KM, two situations may be joined by only one action. Concurrency is thus not a possibility. It has however been introduced in the version resulting from this thesis and is covered in chapter 8. Apart from basic changes to the

system, allowing more than action to join two situations requires handling conflicts between actions. However, actions are performed using the same commands as before.

Concurrency in the form presented above somewhat extends modelling flexibility. However, actions being performed during several situations and actions having delayed effects are still not easily represented. Chapter 9 shows in what way KM has been modified to facilitate this kind of modelling.

[San97a], and to some degree [San97b], discuss success and failure of actions as well as sequences of actions. It has motivated investigating how to model this in KM, the conclusion being that no major changes are needed and that, in fact, KM allows for more kinds of sequences than the ones presented in [San97a]. What has been found out is covered in chapter 10.

The last of the implemented extensions was inspired by a discussion on whether there would be a point in organising situations on different levels of detail, in other words letting the values of instance-slot combinations sometimes vary within situations. What has been concluded is covered in chapter 11.

The last of the presented extensions, continuous change, has not been implemented. After looking into the issue it was concluded that trying to implement continuous change would not be worth the effort when put in relation to other possible extensions.

Table 3.1 aims to show what aspects of the system have been affected by the implemented extensions: querying ("Out"), knowledge base construction ("In"), or both.

---

<i>Extensions</i>	<i>"Out"</i>	<i>"In"</i>
Postdiction	X	
Check of effects and preconditions	X	
Changes to action definitions affecting instance-slot-value relations		X
Non-deterministic direct effects	X	X
Concurrent actions	X	X
New way of modelling actions	X	X
Failed actions and sequences of actions	X	X
Situation levels		X

Table 3.1: Extensions



## Chapter 4

# Postdiction

### 4.1 Inspiration

The idea of introducing postdiction is based mainly on the KM situations manual [CPb], and more specifically on the limitations described in it, which are discussed in section 3.1.

### 4.2 Implementation

Recall that when calculating slot values the basic system takes into account values from earlier situations, but only if those values are compatible with the values already found for the slot in the case of single-valued slots. The approach taken here is similar with the addition that postdicted values are also taken into consideration. As mentioned in section 3.2, values are fetched from a future situation ("postdicted") if the preceding action does not affect them.

When introducing postdiction, if care had not been taken, projection and postdiction could have interacted in such a way that situations would have been visited more than once. To avoid this the system has to keep track of which situations have already been checked in the current query. Using the notation of section 2.3, the new way of computing slot values can

be expressed in the following way:

$$\begin{aligned}
VALUES &= \{x \mid \text{holds-in2}(\text{slot}(\text{instance}, x), \text{sit}, \text{vis-sit})\} \\
&= \text{unify-if-sv}(\text{LOCAL-VALUES}, \text{slot}) \cup_{KM} \\
&\quad \text{unify-if-sv}(\text{SUBSLOT-VALUES}, \text{slot}) \cup_{KM} \\
&\quad \text{unify-if-sv}(\text{INHERITED-VALUES}, \text{slot}) \cup_{KM} \\
&\quad \{y \mid y \in \text{PROJECTED-VALUES} \wedge \neg \text{single-valued}(\text{slot})\} \cup_{KM} \\
&\quad \{z \mid z \in \text{POSTDICTED-VALUES} \wedge \neg \text{single-valued}(\text{slot})\}
\end{aligned}$$

where

$$\begin{aligned}
PROJECTED-VALUES &= \\
&\text{unify-if-sv}(\{x \mid \text{fluent-status}(\text{slot}, *Inertial-Fluent) \wedge \\
&\quad \text{sit} \neq *Global \wedge \\
&\quad \text{prev-situation}(\text{sit}, \text{sit}') \wedge \\
&\quad \text{sit}' \notin \text{vis-sit} \wedge \\
&\quad \text{holds-in2}(\text{slot}(\text{instance}, x), \text{sit}', \text{vis-sit} \cup \text{sit}')\})
\end{aligned}$$

$$\begin{aligned}
POSTDICTED-VALUES &= \\
&\text{unify-if-sv}(\{x \mid \text{fluent-status}(\text{slot}, *Inertial-Fluent) \wedge \\
&\quad \text{sit} \neq *Global \wedge \\
&\quad \text{next-situation}(\text{sit}, \text{sit}') \wedge \\
&\quad \text{sit}' \notin \text{vis-sit} \wedge \\
&\quad \neg \text{affected-by-action}(\text{instance}, \text{slot}, \text{sit}) \wedge \\
&\quad \text{holds-in2}(\text{slot}(\text{instance}, x), \text{sit}', \text{vis-sit} \cup \text{sit}')\})
\end{aligned}$$

and

$$\begin{aligned}
&[\neg \text{single-valued}(\text{slot}) \rightarrow \text{ANS} = \text{VALUES}] \wedge \\
&[\text{single-valued}(\text{slot}) \wedge \\
&\quad \text{unifiable}(\text{VALUES}, \text{PROJECTED-VALUES}, \text{POSTDICTED-VALUES}) \\
&\quad \rightarrow \text{ANS} = \text{VALUES} \cup_{KM} \\
&\quad \quad \text{PROJECTED-VALUES} \cup_{KM} \\
&\quad \quad \text{POSTDICTED-VALUES}] \wedge \\
&[\text{single-valued}(\text{slot}) \wedge \\
&\quad \neg \text{unifiable}(\text{VALUES}, \text{PROJECTED-VALUES}, \\
&\quad \quad \text{POSTDICTED-VALUES}) \\
&\quad \rightarrow \text{ANS} = \text{VALUES}]
\end{aligned}$$



*LOCAL-VALUES*, *SUBSLOT-VALUES*, and *INHERITED-VALUES* are the same as before. *holds-in2(prop, sit, vis-sit)* is a variant of *holds-in*, which takes into account what situations have already been visited. As expected, *next-situation(sit, sit')* is true if and only if *sit'* is the next situation of *sit*. *affected-by-action(instance, slot, sit)* holds if and only if the instance-slot combination is affected by the action performed in situation *sit*. Note that if the slot is single-valued and the projected and postdicted values cannot be unified with *VALUES* neither projected nor postdicted values are used.

### 4.3 Usage

The example from section 2.2, with the addition of the *\*Inertial-Fluent* slot *passengers*, may be used to show in what way postdiction works. Suppose that situation mode is entered and that the *next-situation* command is passed before performing a *Paint* action:

```
(new-situation)
(next-situation)
(do-and-next (a Paint with
              (object (*The-car))
              (action-colours (*Green))))
```

At this point, the car is asserted to carry two passengers:

```
(*The-car has
  (passengers (*Lisa *Kalle)))
```

There are now three situations, named *\_Situation1*, *\_Situation2* and *\_Situation4* due to the way KM numbers anonymous instances. In *\_Situation4*, the car is green and carries the passengers Lisa and Kalle. If a query is passed for the *passengers* slot in *\_Situation1* the system postdicts the values *\*Lisa* and *\*Kalle* from *\_Situation4* through *\_Situation2* to *\_Situation1*. On the other hand, no value is returned for the *colours* slot since its value in *\_Situation4* is a result of the *Paint* action. This reflects the assumption that what is true in a certain situation is true in earlier situations as well provided that no action has made it true.



## Chapter 5

# Checking effects and preconditions

### 5.1 Inspiration and implementation

As mentioned in section 3.1, the original version of KM is not able to detect contradictions between instance-slot-value relations on the one hand and the effects and preconditions of actions on the other. Contradictions related to effects and preconditions arise for four different reasons:

- The add list contains instance-slot-value relations, which do not hold in the next situation.
- The delete list contains instance-slot-value relations, which do hold in the next situation.
- The preconditions list contains instance-slot-value relations, which do not hold in the preceding situation.
- The negated preconditions list contains instance-slot-value relations, which do hold in the preceding situation.

As described in section 2.3, slot value computations in KM are only performed if requested, i.e. when queries are passed either directly from the

user or as part of other computations. Therefore, checking of effects and preconditions should be done in connection to querying.

In the version of KM resulting from the work done on this thesis, contradictions found during checking of effects and preconditions are handled simply by passing error messages. The values of the instance-slot combination passed to the query are not affected. One could imagine changing the values in accordance with the actions in order to eliminate the contradictions, either automatically or after asking the user. However, suppose that the slot values found by a query are first checked against the effects of the preceding action. Contradictions are found but are eliminated by changing the slot values. Next, the system checks the preconditions of the next action and discovers contradictions. In such a case the latter contradictions could very well have been caused by the changes done to the slot values. They could be removed in the same way as the first contradictions but then they could reappear since there actually might be no set of slot values consistent with both the effects of the preceding action and the preconditions of the next action. Therefore, the choice has been made to only let the system produce error messages when contradictions are detected and leave it to the user to change the knowledge base manually. It should also be pointed out that when contradictions arise it is not obvious whether slot values, actions, or both should be changed.

It cannot be expected always to be desirable or necessary for the system to search for contradictions related to the effects and preconditions of actions. One might want to be able to switch off the checking mechanism. This is done by passing the command (`dont-check-effects-and-pcs`). The checking mechanism can be switched on again using the command (`check-effects-and-pcs`). `*Non-Fluent` slots are always excluded from checking of effects and preconditions since their values are global, i.e. do not vary between situations.

## 5.2 Usage

Recall the example from section 2.2. It is used here to demonstrate the checking of effects and preconditions. After entering situation mode a `Paint` action is performed which adds `*Green` to the `colours` slot of `*The-`

car:

```
(do-and-next (a Paint with
              (object (*The-car))
              (action-colours (*Green))))
```

The next step is to execute a Repaint action:

```
(do-and-next (a Repaint with
              (object (*The-car))
              (action-colours (*Yellow *Purple))))
```

However, Repaint now has a slightly different definition containing preconditions and negated preconditions lists:

```
(every Repaint has
  (object ((a Car)))
  (action-colours ((a Colour)))
  (pcs-list (:(triple
              (the object of Self)
              speed
              0)))
  (ncs-list (:(triple
              (the object of Self)
              driving
              *True)))
  (del-list (:(triple
              (the object of Self)
              colours
              (the colours of (the object of Self))))))
  (add-list (:(triple
              (the object of Self)
              colours
              (the action-colours of Self))))))
```

The new lists may be regarded as representing the need for the car not to move while being repainted. There are now three situations and two actions:

```
_Situation1  
_Paint2  
_Situation3  
_Repaint4  
_Situation5
```

Now, the `colours` slot of `*The-car` is changed to only include `*Yellow`:

```
(*The-car now-has (colours (*Yellow)))
```

The `now-has` command has not been explained earlier. It sets the value associated with a certain instance-slot combination and removes all old values as opposed to the `has` command, which merely adds values. A query for the `colours` slot of `*The-car` in `_Situation5` will now produce an error message stating that the values (in this case "value") found for the instance-slot combination are incompatible with the effects of the preceding action `_Repaint4`. The error message is triggered by the absence of `*Purple`.

Apart from adding the values `*Yellow` and `*Purple` to the `colours` slot `_Repaint4` removes the value `*Green` present in `_Situation3`. Therefore, a query for the `colours` of `*The-car` in `_Situation5` will produce an error message if `*Green` has been added to the `colours` slot.

Since the `Repaint` action is performed using the `do-and-next` command the contents of the preconditions list are asserted to be true, i.e. the instance-slot-value relation (`*The-car speed 0`) is asserted to hold. Correspondingly, (`*The-car driving *True`) is asserted not to hold. Setting the value of `driving` to `*True` or the value of `speed` to something other than zero and then passing queries for the slots will result in error messages stating that the found values are incompatible with the preconditions of the next action `_Repaint4`.

## Chapter 6

# Changes to action definitions affecting instance-slot-value relations

### 6.1 Inspiration

During construction of a knowledge base one might need to make changes to action definitions after the actions have been performed. The reason may be either that some parts of the definitions have been found to be wrong or irrelevant or that something that should be included has been overlooked. There might even be a need to introduce new actions joining situations, which were earlier regarded as having no action between them. In order to satisfy this need, changes to action definitions must be allowed to affect instance-slot-value relations.

## 6.2 Implementation

As described in section 2.2, when an action is performed in the original version of KM a new situation, connected to the current situation, is always created, regardless of whether the action has already been performed or not. There is no possibility of "reperforming" actions. If one wanted to introduce such a possibility while retaining modelling of multiple, possible futures there would be a need to create a new set of **redo** commands (or the like) to separate performing actions from "reperforming" them. In the latter case no new next situations would be created. However, since a starting point for all extensions to KM described in this thesis is to allow each situation to have at most one next situation there is no need to introduce **redo** commands. Instead, the existing **do** commands are used in the new version but work in a somewhat different way.

In the original version, an action is always executed in the current situation. In the new version, however, if the action to be performed carries a situation in its **before-situation** slot it is performed in that situation instead. In other words, actions are performed in their specified before situations if such exist and only if that is not the case in the current situation.

In FOL notation, the "old" way of determining the next situation of an action could be formulated

$$performed-in(a, s) \rightarrow \exists s' next-situation(s, s', a)$$

whilst the new way corresponds to

$$performed-in(a, s) \rightarrow [(\exists s'' before-situation(a, s'') \rightarrow next-situation(s'', s', a)) \wedge (\neg \exists s'' before-situation(a, s'') \rightarrow \exists s' next-situation(s, s', a))]$$

## 6.3 Usage

The example given in section 2.2 is used here to illustrate how to make use of the extension to KM presented in this chapter. After entering into situation mode an instance of the action class **Paint** is created:

(a **Paint** with



```
(object (*The-car))  
(action-colours (*Green)))
```

It may be called `_Paint2`, represents the belief that a certain car is painted green and is performed by passing the command

```
(do-and-next _Paint2)
```

Now suppose that at some later point in time it is found out (or rather believed) that the car was actually painted red. To reflect this, `_Paint2` is changed in the following way:

```
(_Paint2 now-has (action-colours (*Red)))
```

`_Paint2` may now be "reperformed" by using either the `do` or the `do-and-next` command. If `do` is chosen the action is performed and the system then returns to whatever was the current situation whilst in the case of `do-and-next` it ends up in the situation following the action. As before, if `try-do` or `try-do-and-next` are used the action is performed only if its preconditions are satisfied.

In order for changes to action definitions to work properly the fluent status of all slots used must be `*Non-Fluent`. Otherwise, changes made in situations other than the one where the action is performed will have no effect since, in that case, the changes are made locally in the current situation and not globally as is necessary. For instance, in the example presented here `object` and `action-colours` should be `*Non-Fluent`.

Notice that, in the above example, the add and delete lists are not directly changed, only the `object` and `action-colours` slots. There is no point in making direct changes to the add, delete, preconditions or negated preconditions lists of an action instance because their old values are removed when the action is performed and new values are computed from the action class definition. There are two reasons for this. Firstly, if old values were not removed reperforming changed actions would not produce the desired result. Secondly, the chosen method ensures that all instances of a certain class, for example `Paint`, follow the class definitions and do not behave in an unexpected way. Additionally, if the definition of a certain action class is changed then when any instance of the class is performed the new definition is used even if the instance was created before the change.



## Chapter 7

# Non-deterministic direct effects

### 7.1 Inspiration

The situations manual [CPb] describes that the original version of KM explicitly prohibits the direct effects of actions to be disjoint, i.e. non-deterministic. This has been the main inspiration for looking into the question of introducing non-deterministic direct effects in KM.

### 7.2 Implementation

Non-deterministic direct effects are represented using the new built-in class `Set-of-effects`. An instance of this class should contain an add list and a delete list specifying a set of effects. A number of sets of effects are then contained in the `sets-of-effects` slot of an action. `sets-of-effects` is a new built-in slot. If an action is associated with a `sets-of-effects` slot carrying a value it is presumed to carry all its effects in that slot and regular add and delete lists are disregarded.

When the action is performed only what is common to all sets of effects

is asserted in the next situation. When the effects of an action containing non-deterministic direct effects are checked for contradictions to the values of an instance-slot combination found during querying in the next situation an error message is given only if the values from the query are inconsistent with all the sets of effects of the action. If one or more of the sets of effects support the values found by the query the values are regarded to be consistent with the action.

In other words, the system reacts if all non-deterministic "delete values" or none of the non-deterministic "add values" appear in the next situation. Thus, if an action carries non-deterministic add and delete effects an error message is not given if at least one delete value is missing in the next situation and at least one add value is present. However, the missing delete value and the present add value may have their origin in different sets of effects. If that is the case there is really no "valid" set of effects. However, the system will not detect this.

Recall that postdiction, presented in chapter 4, is performed if the next action does not affect the instance-slot combination of the query. Using non-deterministic direct effects an action may affect an instance-slot combination even though performing the action does not result in the instance-slot combination taking on new values. For instance, tossing a regular die (see section 7.3) will result in it showing one of its six sides. However, it cannot be expected to be known in advance what the result of the toss will be. Clearly, what side is shown is affected by the action and the result should not be postdicted to previous situations.

It should be pointed out that all sets of effects should contain the same instance-slot combinations. If an instance-slot combination is left out in some set of effects it will receive the value NIL in the situation following the action and it won't be affected by checking of effects.

The FOL formulation of action performing, presented in section 2.2, is not in itself affected by the extension which is the subject of this chapter. However,  $add-list(a, p)$  now holds if  $p$  appears in the regular add list of the action or if the action contains sets of effects and all sets of effects contain  $p$ . The equivalent holds for  $del-list(a, p)$ .

Suppose an action with non-deterministic direct effects is performed and then the user tries to perform a different action carrying some preconditions. In such a case the situation could arise that the preconditions are

---

not satisfied but would be if one of the sets of effects of the first action were allowed to hold. The user could then be informed and asked whether the suitable set of effects should be used or not. However, this has not been implemented.

## 7.3 Usage

An example is used here to demonstrate non-deterministic direct effects. Imagine a die with six sides numbered one to six and a tossing action causing the die to leave the tosser's hand while making a sound and ending up showing one of its six sides.

There are three slots, `number-up` representing the number on the up-side of the die, `makes-sound` and `in-hand`. The slots may be `*Fluent` or `*Inertial-Fluent`. There is also a `Toss` action class with the following definition:

```
(Toss has
  (superclasses (Action)))

(every Toss has
  (sets-of-effects (
    (a Set-of-effects with
      (add-list (:set
        (:triple *The-die number-up 1)
        (:triple *The-die makes-sound Yes))))
      (del-list (:triple *The-die in-hand Yes))))
    (a Set-of-effects with
      (add-list (:set
        (:triple *The-die number-up 2)
        (:triple *The-die makes-sound Yes))))
      (del-list (:triple *The-die in-hand Yes))))
    (a Set-of-effects with
      (add-list (:set
        (:triple *The-die number-up 3)
        (:triple *The-die makes-sound Yes))))
      (del-list (:triple *The-die in-hand Yes))))
```

```
(a Set-of-effects with
  (add-list ([:set
              (:triple *The-die number-up 4)
              (:triple *The-die makes-sound Yes)))))
(del-list ([:triple *The-die in-hand Yes]))
(a Set-of-effects with
  (add-list ([:set
              (:triple *The-die number-up 5)
              (:triple *The-die makes-sound Yes)))))
(del-list ([:triple *The-die in-hand Yes]))
(a Set-of-effects with
  (add-list ([:set
              (:triple *The-die number-up 6)
              (:triple *The-die makes-sound Yes)))))
(del-list ([:triple *The-die in-hand Yes]))))
```

Note that the effect on `makes-sound` and `in-hand` is the same in all sets of effects. Thus, in the situation resulting from performing a `Toss` action the following holds for `*The-die`:

```
(*The-die has
  (in-hand (<<> Yes)))
(makes-sound (Yes)))
```

Since `number-up` carries different values in the different sets of effects it has no value in the situation following the action. Now, suppose a query is passed for `number-up`. Its value will be `NIL`, i.e. nothing but additionally, an error message will be given stating that this value is incompatible with the effects of the preceding action. If `number-up` is given for instance the value 5 there will no error message when a query is passed since the value is "valid" according to the sets of effects. However, if the value given is instead 7 (or any value other than one to six) an error message will appear. Note that the reason for this is that there is no value present between one and six, not that the value is 7.

## Chapter 8

# Concurrent actions

### 8.1 Inspiration

In some situations, for instance when representing more than one agent, one might like to be able to model actions being performed concurrently, i.e. more than one action joining two situations.

### 8.2 Implementation

In the new version of KM produced as a result of this thesis concurrency for actions has been implemented by allowing two situations to be connected to each other by more than one action. The actions are performed using the same commands as usual. When an action is executed the system checks what instance-slot combinations are affected by it and searches all concurrent actions for effects on the same instance-slot combinations. This results in one set of add effects and one set of delete effects, together representing everything happening to the instance-slot combinations between the two situations. The add and delete effects are then checked for contradictions, i.e. if an instance-slot-value relation occurs in both sets it is removed.

Additionally, if the slot of an instance-slot combination is single-valued and there is more than one add effect affecting the instance-slot combina-

tion those add effects are not taken into account. Instead, the instance-slot combination is given the value NIL. This means that executing of actions is changed *in general*, not only in the case of concurrent actions. In the original version of KM, an action may cause an instance-slot combination with a single-valued slot to take on more than one value. The problem is not detected until a query is passed for the instance-slot combination and the value is then changed to NIL. However, there is obviously no limitation on the number of delete effects affecting a single-valued slot.

In FOL, performing actions in the case where preconditions are asserted and not checked can be expressed in the following way:

$$\begin{aligned}
&\forall s, s', p, a \text{ holds-in}(\text{pcs-list}(a, p), s) \wedge \text{next-situation}(s, s', a) \rightarrow \text{holds-in}(p, s) \\
&\forall s, s', p, a \text{ holds-in}(\text{ncs-list}(a, p), s) \wedge \text{next-situation}(s, s', a) \rightarrow \text{holds-in}(\neg p, s) \\
&\forall s, s', p, a [p \in \text{ADD-PROPOSITIONS}(a, s) \wedge \text{next-situation}(s, s', a) \\
&\quad \rightarrow \text{holds-in}(p, s')] \\
&\forall s, s', p, a [p \in \text{DEL-PROPOSITIONS}(a, s) \wedge \text{next-situation}(s, s', a) \\
&\quad \rightarrow \text{holds-in}(\neg p, s')]
\end{aligned}$$

where

$$\begin{aligned}
&\text{ADD-PROPOSITIONS}(a, s) = \\
&\text{remove-multiple-propositions-for-single-valued-slots}( \\
&\quad \text{remove-contradictions}(\text{ALL-ADD-PROPOSITIONS}(a, s)))
\end{aligned}$$

$$\begin{aligned}
&\text{ALL-ADD-PROPOSITIONS}(a, s) = \\
&\{p | \text{action-affects-instance+slot}(a, \text{instance}, \text{slot}) \wedge \\
&\quad \text{proposition-contains-instance+slot}(p, i, s) \wedge \\
&\quad \text{actions-contain-proposition}(\text{add-list}, s, p)\}
\end{aligned}$$

$$\begin{aligned}
&\text{DEL-PROPOSITIONS}(a, s) = \\
&\{p | \text{action-affects-instance+slot}(a, \text{instance}, \text{slot}) \wedge \\
&\quad \text{proposition-contains-instance+slot}(p, i, s) \wedge \\
&\quad \text{actions-contain-proposition}(\text{del-list}, s, p)\}
\end{aligned}$$

$\text{actions-contain-proposition}(\text{list-type}, s, p)$  is true if and only if at least one of the add or delete lists (depending on *list-type*) of the actions performed in



situation  $s$  contains  $p$ . *remove-multiple-propositions-for-single-valued-slots*, *action-affects-instance+slot*( $a, instance, slot$ ) and *proposition-contains-instance+slot*( $p, i, s$ ) all have intuitive functioning. Action performing with check of preconditions is formulated equivalently.

Recall the FOL formula for postdicted values given in section 4.2. *affected-by-action*( $instance, slot, sit$ ) used in it should now be interpreted as the instance-slot combination being affected by at least one of the actions performed in situation  $sit$ . However, the FOL formulation of the slot-value computation procedure stays the same.

### 8.3 Usage

Recall the car example from section 2.2. In this section, it is expanded to include an owner of the car represented by the **\*Non-Fluent** slot **owner**, and a new **Fix-engine** action class. Additionally, some changes are made to the **Remove-colour** action class:

```
(*The-car has
      (owner (*The-owner)))

(Fix-engine has
      (superclasses (Action)))

(every Fix-engine has
  (object ((a Car)))
  (pcs-list ([:set
              (:triple
               (the object of Self)
               speed
               0)
              (:triple
               (the object of Self)
               engine-broken
               *True))]))
  (ncs-list ([:triple
              (the object of Self)
```

```
        driving
        *True)))
(del-list (:set
  (:triple
    (the object of Self)
    engine-broken
    (the engine-broken of (the object of Self)))
  (:triple
    (the owner of (the object of Self))
    angry
    *True))))
(add-list (:triple
  (the object of Self)
  engine-broken
  *False))))

(every Remove-colour has
  (object ((a Car)))
  (action-colours ((a Colour)))
  (pcs-list (:set
    (:triple
      (the object of Self)
      colours
      (the action-colours of Self))
    (:triple
      (the object of Self)
      speed
      0))))
  (ncs-list (:triple
    (the object of Self)
    driving
    *True)))
  (del-list (:triple
    (the object of Self)
    colours
    (the action-colours of Self))))
```

---

```
(add-list ( (:triple
            (the owner of (the object of Self))
            angry
            *True))))
```

The changes made to `Remove-colour` include two new preconditions, one "normal" and one negated, requiring the car not to be moving when colour is removed, and a new add effect representing that removing colour from the car, for some reason, makes the owner angry. Like `Remove-colour`, `Fix-engine` requires the car to be still. Additionally, the engine must be broken. A `Fix-engine` action causes the engine to be not broken and if the owner is angry the anger is removed.

Now, situation mode is entered and a `Paint` action is performed:

```
(do-and-next (a Paint with
              (object (*The-car))
              (action-colours (*Green))))
```

Next, a `Fix-engine` action is executed:

```
(do (a Fix-engine with
     (object (*The-car))))
```

Since the `do` command is used the preconditions are asserted, not checked and the system does not change to the new situation created by performing the action. Executing a concurrent `Remove-colour` action is done simply by passing the command

```
(try-do (a Remove-colour with
         (object (*The-car))
         (action-colours (*Green))))
```

In this case, the preconditions are checked rather than asserted. Since the car has been painted green and since the preconditions related to the `driving` and `speed` slots hold thanks to the `Fix-engine` action all of the preconditions hold and the action is carried out. There are now three situations and three actions:

```

        _Situation1
        _Paint2
        _Situation3
_fix-engine4 _Remove-colour6
        _Situation5

```

Passing the command (showme \*The-car) now produces the following:

```

(*The-car has
  (instance-of (Car))
  (owner (*The-owner))
  (object-of ( _Paint2
              _Fix-engine4
              _Remove-colour6)))

(in-situation _Situation3
  (*The-car has
    (colours (*Green))
    (driving ((<> *True)))
    (speed (0))
    (engine-broken (*True))))

(in-situation _Situation5
  (*The-car has
    (engine-broken (((<> *True)) && (*False))))
  (colours ((<> *Green))))

```

Suppose the values of colours, driving, and speed are changed in \_Situation3:

```

(*The-car now-has
  (colours (*Blue *Yellow))
  (driving (*True))
  (speed (90)))

```

Now, queries for these slots will produce error messages. The values \*Blue and \*Yellow are incompatible with \_Paint2 and \_Remove-colour6 whilst \*True and 90 contradict the preconditions of \_Fix-engine4 and \_Remove-colour6.

## Chapter 9

# New way of modelling actions

### 9.1 Inspiration

The introduction of concurrent actions, presented in chapter 8 allows for some more modelling flexibility by letting more than one action be performed between two situations. However, representing actions being executed during several situations and actions only partly overlapping each other is still not easily done. [Gus01] deals with this and, more specifically, problems associated with the forms of interactions arising from allowing actions that depend on each other to overlap.

Another topic covered by [Gus01] is delayed effects of actions. Representing actions with delayed effects is somewhat difficult in the original version of KM. It is to some degree possible to let slot values depend on the state of the world in an earlier situation but encoding rules of this kind is rather cumbersome.

## 9.2 Implementation

Imagine a scenario with two action classes, `Light-fire` and `Pour-water`, and an instance `*The-wood` of the class `Wood`:

```
(Light-fire has
  (superclasses (Action)))

(every Light-fire has
  (object ((a Wood)))
  (pcs-list ([:triple
              (the object of Self)
              dry
              *True])))
  (del-list ([:triple
              (the object of Self)
              on-fire
              *False])))
  (add-list ([:triple
              (the object of Self)
              on-fire
              *True]])))

(Pour-water has
  (superclasses (Action)))

(every Pour-water has
  (object ((a Wood)))
  (del-list ([:triple
              (the object of Self)
              dry
              *True])))
  (add-list ([:triple
              (the object of Self)
              dry
              *False]])))
```

```
(*The-wood has
  (instance-of (Wood)))
```

Performing actions of these kinds one after another does not present a problem. However, suppose a `Light-fire` action and a `Pour-water` action are executed simultaneously:

```
(new-situation)
(do (a Light-fire with (object (*The-wood))))
(do-and-next (a Pour-water with (object (*The-wood))))
```

Both `(*The-wood on-fire *True)` and `(*The-wood dry *False)` will now hold, which is clearly unintuitive. A similar problem arises if actions are allowed to have duration, i.e. connect nonadjacent situations, and thus possibly partly overlap each other. Suppose, for instance, that a `Light-fire` action is executed and that a `Pour-water` action is then "started" before the `Light-fire` action has finished, i.e. in a situation between its start and end situations. In that case, the wood will be on fire in the end situation of the `Light-fire` action and be wet (but still on fire) in the end situation of the `Pour-water` action.

This clearly shows that if actions are allowed to be performed simultaneously, either between adjacent or nonadjacent situations, the interactions between them need to be handled. The standard way of modelling actions in KM is to let them directly affect "real" things in the world. For instance, `Light-fire` changes the value of the `on-fire` slot. One could imagine preserving this way of modelling and handle interactions in the action definitions themselves. However, for each action all other actions which could potentially interact with it would have to be included in the action definition. In the case of actions with duration, all possible combinations of overlapping would also have to be covered. The task of representing actions in this way is further complicated if non-deterministic effects are taken into consideration.

[Gus01] presents an alternative to this way of representing actions, which has been applied to KM. Instead of directly affecting what is regarded to be "real" features, i.e. instance-slot combinations, of the world, actions affect what is referred to as "influences". The values of the "real"

features are then determined by rules relating to the influences. They are thus only indirectly affected by the actions.

To apply this to KM, slots are divided into four types:

- The equivalent of influences are directly affected by the actions and their values should be both projected and postdicted. They are therefore represented by **\*Inertial-Fluent** slots.
- Rule-based slots with default values should not have their values projected or postdicted but the values should be allowed to vary between situations. They should therefore be **\*Fluent**.
- In the case of rule-based slots with no default values values should be fetched from earlier situations but only if the rules do not produce a value. Postdiction should never be carried out. The reason for this is that slot values should only be fetched from later situations if the actions leading there do not affect the instance-slot combinations. Since one cannot know whether the value of this kind of slot is the result of actions (through application of the rules) or observations postdiction must be excluded. Since the existing kind of slots do not fit these requirements, a new kind of slot has been introduced, **\*Rule-Based-Inertial-Fluent**.
- As usual, **\*Non-Fluent** slots are used when the slot values should not vary between situations.

In FOL notation, postdiction is now carried out in the following way:

$$\begin{aligned}
 \text{POSTDICTED-VALUES} = & \\
 \text{unify-if-sv}(\{x | & (\text{fluent-status}(\text{slot}, *Inertial-Fluent) \vee \\
 & \text{fluent-status}(\text{slot}, *Rule-Based-Inertial-Fluent)) \wedge \\
 & \text{sit} \neq *Global \wedge \\
 & \text{next-situation}(\text{sit}, \text{sit}') \wedge \\
 & \text{sit}' \notin \text{vis-sit} \wedge \\
 & \neg \text{affected-by-action}(\text{instance}, \text{slot}, \text{sit}) \wedge \\
 & \text{holds-in2}(\text{slot}(\text{instance}, x), \text{sit}', \text{vis-sit} \cup \text{sit}')\})
 \end{aligned}$$

In order to make it easier to represent actions with delayed effects, six new kinds of conditional operators have been introduced:



(`in-all-n-prev-situations n <expr>`) is true if `expr` is true in all the `n` situations preceding the current situation.

(`in-curr-and-all-n-prev-situations n <expr>`) works like `in-all-n-prev-situations` except that the current situation is also checked.

(`in-all-n-prev-situations-m-situations-back n m <expr>`) works like `in-all-n-prev-situations` except that the system starts checking situations `m` situations back in time.

(`in-any-of-n-prev-situations n <expr>`) checks the same situations as `in-all-n-prev-situations` but returns true if `expr` is true in *any* of them.

(`in-curr-or-any-of-n-prev-situations n <expr>`) works like `in-any-of-n-prev-situations` but first checks the current situation.

(`in-any-of-n-prev-situations-m-situations-back n m <expr>`) first moves back `m` situations.

## 9.3 Usage

The fire example presented in section 9.2 is used to illustrate the new way of modelling actions. There are now two "real" kinds of slots and two influence slots:

```
(on-fire has
  (instance-of (Slot))
  (fluent-status (*Rule-Based-Inertial-Fluent))
  (cardinality (N-to-1)))
```

```
(dry has
  (instance-of (Slot))
  (fluent-status (*Fluent))
  (cardinality (N-to-1)))
```

```
(lighting-fire has
  (instance-of (Slot))
  (fluent-status (*Inertial-Fluent))
  (cardinality (N-to-1)))
```

```
(pouring-water has
  (instance-of (Slot))
  (fluent-status (*Inertial-Fluent))
  (cardinality (N-to-1)))
```

There are two kinds of actions, a lighting action and a pouring action, represented by one start event and one stop event each. The instance used for handling the influence slots in the example is called *\*Things-going-on* but this is not a built-in name. Instead, the user may decide how to handle the influence slots.

```
(Start-lighting-fire has
  (superclasses (Event)))
```

```
(every Start-lighting-fire has
  (del-list ( (:triple
               *Things-going-on
               lighting-fire
               (the lighting-fire of *Things-going-on))))
  (add-list ( (:triple
               *Things-going-on
               lighting-fire
               *True))))
```

```
(Stop-lighting-fire has
  (superclasses (Event)))
```

```
(every Stop-lighting-fire has
  (del-list ( (:triple
               *Things-going-on
               lighting-fire
               (the lighting-fire of *Things-going-on))))
```

---

```
(add-list ( (:triple
             *Things-going-on
             lighting-fire
             *False))))

(Start-pouring-water has
  (superclasses (Event)))

(every Start-pouring-water has
  (del-list ( (:triple
               *Things-going-on
               pouring-water
               (the pouring-water of *Things-going-on))))
  (add-list ( (:triple
               *Things-going-on
               pouring-water
               *True))))

(Stop-pouring-water has
  (superclasses (Event)))

(every Stop-pouring-water has
  (del-list ( (:triple
               *Things-going-on
               pouring-water
               (the pouring-water of *Things-going-on))))
  (add-list ( (:triple
               *Things-going-on
               pouring-water
               *False))))

There is also an object *The-wood:

(*The-wood has
  (dry ((if ((in-curr-or-any-of-n-prev-situations
              3 ((the pouring-water of *Things-going-on)
                 = *True))))
```

```
        then *False
        else *True)))
(on-fire ((if (((the lighting-fire of *Things-going-on)
               = *True)
              and ((the dry of Self) = *True))
          then *True
          else (if (((the dry of Self) = *False))
                  then *False))))))
```

The rules contained in the slots of *\*The-wood* signify that the wood is dry if water has not been poured on it for the last three situations and that the wood is on fire if it is dry and a lighting action is going on. The wood is explicitly not on fire if the wood is wet.

Imagine the following simple case:

```
(do-and-next (a Start-lighting-fire))
(do-and-next (a Stop-lighting-fire))
```

Now, the *on-fire* slot of *\*The-wood* will have *\*True* in all situations except the first where the value will be unknown. Notice that *dry* will always have a value in contrast to *on-fire*.

Introducing a pouring action will complicate things somewhat:

```
(do-and-next (a Start-lighting-fire))
(do-and-next (a Start-pouring-water))
(do-and-next (a Stop-lighting-fire))
(do-and-next (a Stop-pouring-water))
```

In this case, the wood is never on fire. The same holds if the actions are performed concurrently:

```
(do (a Start-lighting-fire))
(do-and-next (a Start-pouring-water))
(do (a Stop-lighting-fire))
(do-and-next (a Stop-pouring-water))
```

## Chapter 10

# Failed actions and sequences of actions

### 10.1 Inspiration

The introduction of failed actions and sequences of actions has mainly been inspired by [San97a]. The basic idea is that actions may succeed or fail and that this can be used to model sequences of actions where the actions to be performed are not predefined but depend on the outcome of each other. More specifically, there are two kinds of sequences. In the first case, the actions in the sequences are performed one by one until either an action fails or there are no more actions to perform. In the second case, the "stop condition" is success instead of failure, i.e. the actions are tried until an action succeeds or there are no more actions.

### 10.2 Implementation

When failed actions and sequences of actions were first considered for this thesis the approach taken was to introduce a built-in slot `failed` to be used to represent failure of actions. However, it was later realised that failure

and sequences of actions could be modelled without the aid of such a slot, in fact without making major changes to the KM code.

As part of the RKF project mentioned in section 2.5, a new command `do-plan` was introduced in KM. This command is not mentioned in the KM manuals but is included in the system. A plan to be performed using `do-plan` should always be associated with at least two slots, `first-subevent` specifying where to start and `subevent` containing the possible actions of the plan. The `subevent` slot could actually be named something else but there does not seem to be any point in using a different name. Each action of the plan should be associated with a `next-event` slot specifying the next action to be performed. The choice of next action to perform may or may not depend on the outcome of a test specified in a `next-event-test` slot.

The `do-plan` command offers a possibility of modelling not only the mentioned two kinds of action sequences but also loops. Plans can therefore be used to represent "higher-level" actions consisting of common actions performed in a non-predefined order. An example of this is given in section 10.3. Unfortunately, the original version of KM does not allow the parts of a plan to be plans in themselves but only actions. However, this has been changed in the new version, which means that it is possible to combine plans to form higher-level plans.

Even though the `failed` slot is not required in order to model sequences of actions it has been included and does serve a purpose. It is imaginable that one would like to be able to replace one action with another but still be able to change back to the old one. In other words, one might want to link several actions to a pair of situations but only have the system consider one or some of them at a time. This is what happens automatically if `*permit-concurrent-actions*` is set to nil.

## 10.3 Usage

The following example has been inspired by [blo05]. Imagine a tower consisting of blocks which should be unstacked. Only one block, the one on top, may be removed from the tower at a time. The fluent status of the following slots is other than `*Fluent*`:

```
(instrument has (fluent-status (*Non-Fluent)))
```

---

```
(object has (fluent-status (*Non-Fluent)))
(tower has (fluent-status (*Non-Fluent)))
(plan has (fluent-status (*Non-Fluent)))
(parts has (fluent-status (*Rule-Based-Inertial-Fluent)))
(on has (fluent-status (*Rule-Based-Inertial-Fluent)))
(clear? has (fluent-status (*Rule-Based-Inertial-Fluent)))
```

There is a Remove action class and an empty Finish action class:

```
(Remove has
  (superclasses (Action)))

(every Remove has
  (object ((a Block)))
  (tower ((a Tower)))
  (pcs-list (:triple
    (the object of Self)
    clear?
    Yes)))
  (del-list (:triple
    (the object of Self)
    on
    (the on of (the object of Self)))
    (:triple
    (the tower of Self)
    parts
    (the object of Self))
    (:triple
    (the object of Self)
    parts-of
    (the parts-of of (the object of Self)))
    (:triple
    (the on of (the object of Self))
    clear?
    No)))
  (add-list (:triple
    (the on of (the object of Self))
```

```

        clear?
        Yes)))
(:triple
 (the object of Self)
 on
 *Floor))))

```

```

(Finish has
 (superclasses (Action)))

```

There is also an Unstack-plan plan class:

```

(Unstack-plan has
 (superclasses (Plan)))

```

```

(every Unstack-plan has
 (object ((a Tower)))
 (first-subevent ((the subevent of Self) called "remove")))
 (subevent ((a Remove called "remove" with
 (object ((the Block with
 (parts-of ((the Tower object of Self)))
 (clear? (Yes))))))
 (tower ((the Tower object of Self)))
 (next-event-test
 ('((the number of (the parts of
 (the Tower object of Self))) = 1)))
 (next-event ([:args t ((the subevent of Self)
 called "finish"))
 (:args NIL ((the subevent of Self)
 called "remove")))))
 (a Finish called "finish"))))

```

Suppose there are four blocks arranged in a tower. This scenario is set up in the following way:

```

(*MyTower has
 (instance-of (Tower))
 (parts (*BlockA *BlockB *BlockC *BlockD)))

```



```
(*BlockA has
  (instance-of (Block))
  (on (*BlockB))
  (clear? (Yes)))
```

```
(*BlockB has
  (instance-of (Block))
  (on (*BlockC))
  (clear? (No)))
```

```
(*BlockC has
  (instance-of (Block))
  (on (*BlockD))
  (clear? (No)))
```

```
(*BlockD has
  (instance-of (Block))
  (on (*Floor))
  (clear? (No)))
```

An Unstack-plan is executed by passing the command

```
(do-plan (a Unstack-plan with (object (*MyTower))))
```

Passing a (showme \*MyTower) command now illustrates what the plan has achieved:

```
(*MyTower has
  (instance-of (Tower)))
```

```
(in-situation _Situation1
  (*MyTower has
    (parts (*BlockA
            *BlockB
            *BlockC
            *BlockD))
    (object-of (_Unstack-plan2))
    (tower-of (_Remove3))))
```

```

(in-situation _Situation5
  (*MyTower has
    (parts (*BlockB
            *BlockC
            *BlockD
            (<> *BlockA)))
    (object-of (_Unstack-plan2))
    (tower-of (_Remove6))))

(in-situation _Situation9
  (*MyTower has
    (parts (*BlockC
            *BlockD
            (<> *BlockB)))
    (object-of (_Unstack-plan2))
    (tower-of (_Remove10))))

(in-situation _Situation11
  (*MyTower has
    (parts (*BlockD
            (<> *BlockC)))
    (object-of (_Unstack-plan2))))

```

**Remove** actions are thus performed until only one block remains in the tower. Whether a **Remove** action should be considered to have failed if no blocks are stacked on each other any longer or if blocks are still stacked on each other is just a matter of perspective. In the former case the "stop condition" of the sequence above would be failure whilst in the second case it would be success. However, the modelling is the same.

Note that three of the declared slots are **\*Rule-Based-Inertial-Fluent**. However, they are not used in the way described in chapter 9. The reason for making them **\*Rule-Based-Inertial-Fluent** is instead that allowing postdiction causes looping in the case of plans.

As mentioned in section 10.2, a minor change has been made to KM to allow the parts of a plan to be plans themselves. Recall the fire example from section 9.3. It is expanded here to include an empty **Continue** event class and three plan classes:

---

```
(Continue has (superclasses (Event)))

(Light-fire-plan has (superclasses (Plan)))

(every Light-fire-plan has
  (object ((a Wood)))
  (first-subevent (((the subevent of Self) called "start lighting")))
  (subevent ((a Start-lighting-fire called "start lighting" with
    (next-event-test (
      '((the on-fire of (the object of Self)) = *True)))
    (next-event (
      (:args t ((the subevent of Self)
        called "stop lighting"))
      (:args NIL ((the subevent of Self)
        called "continue")))))
    (a Continue called "continue" with
      (next-event-test (
        '((the on-fire of (the object of Self)) = *True)))
      (next-event (
        (:args t ((the subevent of Self)
          called "stop lighting"))
        (:args NIL ((the subevent of Self)
          called "continue")))))
      (a Stop-lighting-fire called "stop lighting")))))

(Extinguish-fire-plan has (superclasses (Plan)))

(every Extinguish-fire-plan has
  (object ((a Wood)))
  (first-subevent (((the subevent of Self) called "start pouring")))
  (subevent ((a Start-pouring-water called "start pouring" with
    (next-event-test (
      '((the on-fire of (the object of Self)) = *False)))
    (next-event (
      (:args t ((the subevent of Self)
        called "stop pouring"))
      (:args NIL ((the subevent of Self)
```

```

                                called "continue")))))
(a Continue called "continue" with
  (next-event-test (
    '((the on-fire of (the object of Self))
      = *False)))
  (next-event (
    (:args t ((the subevent of Self)
              called "stop pouring"))
    (:args NIL ((the subevent of Self)
                 called "continue")))))
(a Stop-pouring-water called "stop pouring"))))

(Light-and-extinguish-fire-plan has (superclasses (Plan)))

(every Light-and-extinguish-fire-plan has
  (object ((a Wood)))
  (first-subevent ((the subevent of Self) called "light fire")))
  (subevent ((a Light-fire-plan called "light fire" with
    (object ((the object of Self)))
    (next-event ((the subevent of Self)
                 called "extinguish fire"))))
  (a Extinguish-fire-plan called "extinguish fire" with
    (object ((the object of Self))))))

```

First, water is poured on the wood:

```

(do-and-next (a Start-pouring-water))
(do-and-next (a Stop-pouring-water))

```

Next, a Light-and-extinguish-fire-plan is executed:

```

(do-plan (a Light-and-extinguish-fire-plan with (object (*The-wood))))

```

The water pouring will result in the wood first being dry and then wet for three situations. The Light-and-extinguish-fire-plan is started immediately after the Stop-pouring-water action, its first part being a Light-fire-plan, but does not result in the wood being on fire until it

is once again dry. The lighting "sub-plan" is followed by an **Extinguish-fire-plan** which puts out the fire. Events and situations thus occur in the following order:

\_Situation1  
\_Start-pouring-water2  
\_Situation3  
\_Stop-pouring-water4  
\_Situation5  
\_Start-lighting-fire9  
\_Situation12  
\_Continue14  
\_Situation19  
\_Stop-lighting-fire22  
\_Situation23  
\_Start-pouring-water26  
\_Situation29  
\_Stop-pouring-water32  
\_Situation35



# Chapter 11

## Situation levels

### 11.1 Inspiration and implementation

The extension, or rather extensions, presented in this chapter arose as a result of a discussion on the nature of situations. More specifically, the question was if there would be a point in organising situations on different levels of detail, in other words letting the values of instance-slot combinations sometimes vary within situations. This could be formulated as a situation being *unambiguous* with respect to possibly only some of the instance-slot combinations, namely the ones carrying the same value throughout the situation. Formally, this could be expressed in the following way:

- The value of a feature (instance-slot combination)  $f$  at time point  $t$  is  $u(f, t)$ .
- A trajectory on  $[t_1, t_2]$  for  $f$  is the set of values taken by  $f$  in the interval, i.e.  $\{u(f, t) : t_1 \leq t \leq t_2\}$ .
- A situation, which is unambiguous with respect to  $F = \{f_1, f_2, \dots\}$  is the set of trajectories whose features belong to  $F$  and carry constant values in the interval.

One reason for making such a change would be to let the user "extract" the parts of the knowledge base relating to a certain set of instance-slot combinations while disregarding the rest. Thus, for each combination of instance-slot combinations, a *situation level* could be made in which shifts between situations would occur only when one or more of the included instance-slot combinations changed their values and in which only those instance-slot combinations could be affected. In this way, parts of the knowledge base irrelevant to the user at a certain time could be filtered away. However, this would not achieve increased expressivity. Instead, one would have to adapt actions to situation levels and the use of rules in situation levels would have to be restricted to include only the specified instance-slot combinations. Therefore, situation levels of this kind have not been implemented. However, it has been considered worth implementing a different kind, whose purpose is simply to show in what way a set of instance-slot combinations vary and thus not to create partitions of the knowledge base. Such a situation level is created by passing the command

```
(make-situation-level ((<instance1> <slot1-1> <slot1-2> ...)
                     (<instance2> <slot2-1> <slot2-2> ...)
                     ...))
```

The system will then divide all "normal" situations into groups, the values of the provided instance-slot combinations being constant within each group. For each group, a "joining situation" is created with the included normal situations being specified in a `joined` slot. The joining situations of a situation level, contained in the `contained-situations` slot, are thus unambiguous with respect to all the provided instance-slot combinations.

Joining situations are distinguished from normal situations by carrying the value `*True` in the slot `joining` and differ from the normal ones by not being connected to other situations and thus not affecting slot-value computation. They can be produced not only as parts of situation levels but also by passing the command

```
(make-joining-situation (<situation1> <situation2> ...)
                       ((<instance1> <slot1-1> <slot1-2> ...)
                        (<instance2> <slot2-1> <slot2-2> ...)
                        ...))
```



The purpose of creating joining situations in this way is to easily find out what instance-slot-value relations are common to a set of situations. Passing the `make-joining-situation` command will make the system check which of the instance-slot combinations have constant value in the situations. The joining situation is thus unambiguous with respect to these combinations. This is represented in the situation by the slot `unambiguous-wrt`. The provided situations are "stored" in the slot `joined-situations`. Since the system only checks the provided instance-slot combinations there is a possibility that a joining situation is actually unambiguous with respect to more combinations than the ones included in the `unambiguous-wrt` slot. In principle, it would be possible to have the `make-joining-situation` command check all instance-slot combinations in the knowledge base. However, this would give rise to extensive computations and it may be presumed that the user has no or little interest in other instance-slot combinations than the ones provided. Since joining situations do not affect slot-value computation any class could be used to represent them. However, by using the `Situation` class `make-joining-situation` works without any adjustment for joining situations as well as normal ones.

When creating a situation level for a certain set of instance-slot combinations for which a situation level already exists the system by default overwrites the old situation level, i.e. changes the values contained in its slots. Thus, no new instance of the `Situation-level` class is created. However, the default setting can be changed by using the command `(dont-overwrite-situation-levels)` which sets the value of the global variable `*overwrite-situation-levels*` to `NIL`. This causes the system to create new instances of `Situation-level` each time `make-situation-level` is passed. The value of `*overwrite-situation-levels*` may be changed back to `t` by using `(overwrite-situation-levels)`.

## 11.2 Usage

Suppose there is an instance `*Kalle`, associated with five slots, and nine situations. The slot values vary between the situations as shown in table 11.1, T signifying `*True` and F signifying `*False`. Making a joining situation for situation 1, 2, and 3 with respect to all the slots in `*Kalle` is

---

Situation	hungry	cold	happy	tired	old
1	T	T	T	T	T
2	T	T	T	<b>F</b>	T
3	T	T	<b>F</b>	T	T
4	T	<b>F</b>	<b>F</b>	<b>F</b>	T
5	T	<b>F</b>	T	T	T
6	T	<b>F</b>	T	<b>F</b>	T
7	T	T	<b>F</b>	T	T
8	T	T	<b>F</b>	<b>F</b>	T
9	T	T	T	T	T

Table 11.1: Slot values

done by passing the following command:

```
(make-joining-situation (_Situation1 _Situation2 _Situation3)
  ((*Kalle hungry cold happy tired old)))
```

This produces the following message:

```
The situations (_Situation1 _Situation2 _Situation3) have been
joined in situation _Situation10. The joining situation is
unambiguous with respect to (at least) the following:
```

```
((:triple *Kalle hungry (:set *True))
 (:triple *Kalle cold (:set *True))
 (:triple *Kalle old (:set *True)))
```

As expected, the created situation is unambiguous with respect to **hungry**, **cold**, and **old**, which carry the same value in situations 1, 2, and 3. If all nine situations are included instead the joining situation will be unambiguous with respect to only **hungry** and **old**.

A situation level for, for instance, **cold** and **happy** is made using this command:

```
(make-situation-level ((*Kalle cold) (*Kalle happy)))
```

This will give rise to six joining situations encompassing situations 1 and 2, 3, 4, 5 and 6, 7 and 8, and 9, respectively as can be deduced from table 11.1.

## Chapter 12

# Results and future work

This thesis has resulted in a new version of the KM system that can handle some new kinds of tasks related to reasoning about actions and change. However, the new version does not include simulation of multiple, possible futures. Summarising, the features introduced in the version resulting from this thesis include the following:

- Instance-slot combinations whose slots are **\*Inertial-Fluent** now have their values projected backwards (postdicted) from one situation to the previous provided that the action connecting the situations does not affect the instance-slot combination.
- The system can now alert the user during a slot-value computation if the values found for the instance-slot combination are incompatible with the effects of the preceding action(s) or the preconditions of the next action(s).
- Changes made to actions after they have been performed may now affect instance-slot-value relations.
- It is now possible to model non-deterministic direct effects of actions.
- In the new version, two situations may be connected by more than one action, i.e. concurrent actions are allowed.

- 
- Actions being performed during several situations, actions partly overlapping each other, and actions with delayed effects are much more easily modelled.
  - Thanks to the introduction of situation levels and joining situations, the new version provides an efficient means of determining in what way instance-slot-value relations vary as well as what instance-slot-value relations are common to a set of situations.

Additionally, the thesis shows that modelling failure and sequences of actions can be done (mainly) within the original version of KM.

As described throughout the thesis, some possible extensions have been thought of but not implemented. These could be the subject of future work and they include the following:

- Continuous change (chapter 3)
- When performing an action in a situation following an action with non-deterministic direct effects the preconditions of the action to be performed might only be satisfied if one (or some) of the sets of effects of the non-deterministic action are allowed to hold. In such cases, the user could be notified and given the choice to let the designated set of effects hold (section 7.2).
- Making sure the check of effects reacts if an action carrying non-deterministic direct effects has no "valid" set of effects (section 7.2).
- Introducing postdiction for **\*Rule-Based-Inertial-Fluent** slots, thus eliminating the need for this special kind of slot (section 9.2).

# Appendix A

## Dictionary

**Action** An action connects two situations and may cause the values of certain instance-slot combinations to change. It may also carry pre-conditions.

**Botany Knowledge Base project** It was as part of this project that the first version of KM was created.

**Cardinality** A slot may carry a certain cardinality, describing in what way it can connect frames to each other.

**Chronological minimization** A limitation of KM described in the situations manual [CPb].

**Class** A kind of frame representing a group of individuals sharing some properties.

**Continuous change** A limitation of KM described in the situations manual [CPb].

**Disjunctive ramifications** A limitation of KM described in the situations manual [CPb], the same thing as non-deterministic indirect effects.

---

**Domain** The domain of a slot is the most general class(es) allowed for an instance using the slot.

**Fluent status** The fluent status of a slot determines in what way its values should persist between situations.

**Frame** Frames are the basic unit of representation in KM. They may be either instances or classes.

**Halo project** An ongoing project in which KM is used by one of the participating teams.

**Instance** A kind of frame. An instance may be anonymous or named.

**Inverse** Every slot has an inverse, representing the opposite relation. The default inverse of *slot* is *slot-of*.

**Lisp** The programming language used for implementing KM.

**Non-deterministic effects** Effects with unknown outcome. They may be direct or indirect.

**Postdiction** The idea that facts known to hold in the future may be presumed to hold at the current point in time provided they have not been made to hold "on the way".

**Projection** The idea that facts known to hold earlier may be presumed to hold at the current point in time.

**Range** The range of a slot is the most general class(es) allowed for its filler.

**Rapid Knowledge Formation project** A project that was conducted from 1999 to 2003 in which one of the participating teams used KM.

**Situation** The main concept of KM's situation mechanism.

**Situation-action dichotomy** One limitation of KM described in the situations manual [CPb].

**Slot** Slots are used in KM to represent relations between frames.

**Slot value** A value "contained" by a slot.

**Subclass** The subclass of a class represents a subset of its members, i.e. the subclass is more specific than the original class.

**Subslot** Subslots relate to slots in the same way as subclasses relate to classes.

**Superclass** The superclass of a class represents a superset of its members, i.e. the superclass is more general than the original class.

**Superslot** Superslots relate to slots in the same way as superclasses relate to classes.

**Variable** Variables can be used in KM to make it easier to formulate expressions.





## Appendix B

# New concepts in KM

In order of appearance in the thesis:


- (check-effects-and-pcs)
- (dont-check-effects-and-pcs)
- Set-of-effects
- sets-of-effects
- \*Rule-Based-Inertial-Fluent
- (in-all-n-prev-situations n <expr>)
- (in-curr-and-all-n-prev-situations n <expr>)
- (in-all-n-prev-situations-m-situations-back n <expr>)
- (in-any-of-n-prev-situations n <expr>)
- (in-curr-or-any-of-n-prev-situations n <expr>)
- (in-any-of-n-prev-situations-m-situations-back n <expr>)
- failed

- 
- (permit-concurrent-actions)
  - (dont-permit-concurrent-actions)
  - (make-situation-level <instances+slots>)
  - joined
  - contained-situations
  - joining
  - (make-joining-situation <situations> <instances+slots>)
  - unambiguous-wrt
  - joined-situations

# Bibliography

- [bkb05] Botany knowledge base project home page. <http://www.cs.utexas.edu/users/mfkb/RKF/projects/bkb.html>, January 2005.
- [blo05] Blocks example. <http://www.cs.utexas.edu/users/pclark/rkf/sadl/blocks.km>, January 2005.
- [CPa] Peter Clark and Bruce Porter. Km - the knowledge machine - inference engine 2.0.8.
- [CPb] Peter Clark and Bruce Porter. *KM - The Knowledge Machine 1.4.0: KM's Situation Mechanism*.
- [CPc] Peter Clark and Bruce Porter. *KM - The Knowledge Machine 2.0: Users Manual*.
- [Gus01] Joakim Gustafsson. Extending temporal action logic, April 2001.
- [hal05] Halo project home page. <http://www.projecthalo.com/>, January 2005.
- [RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.
- [San97a] Erik Sandewall. Logic-based modelling of goal-directed behavior. In *Electronic Transactions on Artificial Intelligence*, volume 1, pages 105–128. 1997.

- 
- [San97b] Erik Sandewall. Relating high-level and low-level action descriptions in a logic of actions and change, 1997.
- [sri05] Sri team home page. <http://www.ai.sri.com/project/SHAKEN>, January 2005.

 <b>Avdelning, Institution</b> Division, Department  AIICS, Dept. of Computer and Information Science 581 83 Linköping		<b>Datum</b> Date  2005-03-14
<b>Språk</b> Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English  <input type="checkbox"/> _____	<b>Rapporttyp</b> Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	<b>ISBN</b> — <hr/> <b>ISRN</b> LITH-IDA-EX-05/026-SE <hr/> <b>Serietitel och serienummer ISSN</b> Title of series, numbering _____
<b>URL för elektronisk version</b>  <a href="http://www.ep.liu.se/exjobb/ida/2005/dd-d/026/">http://www.ep.liu.se/exjobb/ida/2005/dd-d/026/</a>		
<b>Titel</b> Utökning av The Knowledge Machine  Title Extending the Knowledge Machine  <b>Författare</b> Markus Ingevall Author		
<b>Sammanfattning</b> Abstract  <p style="text-align: center;">This master's thesis deals with a frame-based knowledge representation language and system called The Knowledge Machine (KM), developed by Peter Clark and Bruce Porter at the University of Texas at Austin. The purpose of the thesis is to show a number of ways of changing and extending KM to handle larger classes of reasoning tasks associated with reasoning about actions and change.</p>		
<b>Nyckelord</b> Knowledge representation, reasoning about actions and change, The Keywords Knowledge Machine, KM		



# Copyright

## Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om *Linköping University Electronic Press* se förlagets hemsida <http://www.ep.liu.se/>

## English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the *Linköping University Electronic Press* and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>