# KM – The Knowledge Machine 1.4.0: KM's Situation Mechanism

(Revision 1, for KM 1.4.0-beta38 and later.
See release notes for recent updates)

Peter Clark* and Bruce Porter
Dept of Computer Science
University of Texas at Austin, USA
{pclark,porter}@cs.utexas.edu

## Contents



*Current address: Knowledge Systems, Boeing Maths and Computing Technology, PO Box 3707 (m/s 7L66), Seattle, WA 98124 (peter.e.clark@boeing.com)

# 1 Introduction

This manual describes the use of *situations* in KM, providing the ability to represent and reason about states of the world, and actions which change those states. The use of situations is optional in KM – if the user has no need for them, this entire manual can be skipped.

Many tasks require reasoning about the dynamics of the world: that is, require consideration of how a particular situation might change with time. For example, answering questions of the form "what happens if...?", "is it possible that...?", and "how can I...?" all require reasoning about changes in, and alternatives to, the current state of the world. In this manual, we describe how KM represents and reasons with situations. Although this feature of KM was originally motivated to represent world dynamics, it's applicability is more general: it can be used to represent and reason with any hypothetical or possible situation (even if it has no temporal relationship to other situations). For example, KM might be reasoning about a fuel-tank, but not know whether it is full or empty: it can spawn two separate branches or reasoning, one in which the tank is full and the other in which it is empty, and examine the implications of both. Both branches are maintained in KM's memory, and can be explored and compared.

Although powerful, this kind of reasoning also raises some technical and philosophical challenges for AI (for example: the nature of persistence, temporal projection, lines of identity, and multiple, possible futures). We discuss these throughout this manual. First we introduce the notion of *situations*, then describe the representation of *actions*, and how (theories describing) new situations resulting from actions can be computed. We then generalize this to simulating the execution of plans (action networks), and finally to constructing and reasoning about possible, alternative situations which can arise from an initial starting point.

The semantics of KM's situations are based on Situation Calculus (see [1] and [2, Chapter 5]), although there are some differences (for example, our notation differs, and we allow situations to be organized hierarchically).

**Logic Notation:** As in the KM Manual, KM statements in this manual are accompanied by their equivalent in logic notation. However, for those not fluent with logic, do not be intimidated: These logic statements are not essential to decipher, as the accompanying text also provides descriptions of what the KM statements mean. Conversely, for those requiring more rigor, the logic equivalents will help to provide additional clarity.

# 2 Situations

## 2.1 Creating and Entering Situations

The key concept for reasoning about different configurations of the world is a **situation**. Intuitively, a situation describes the state of the world at a particular moment. A situation acquires ('imports') all the axioms (ie. KM frames) from the 'global' (normal) KB, which we refer to as the 'global situation', and can have additional, situation-specific facts asserted into it. Implementation-wise, a situation can be thought of as a partition of the knowledge-base which is (by default) 'isolated' from all other situations except the global (shared) KB. A user can tell KM to 'enter' a situation, in which case future computation is performed only with axioms visible in that situation, and the results stored in that situation. By default, KM works in the global situation (as signified by the `KM>` prompt) – for instance, all the examples in the User Manual are performed in the global situation.

Situations are objects in their own right in the KB (instances of the class `Situation`), and thus

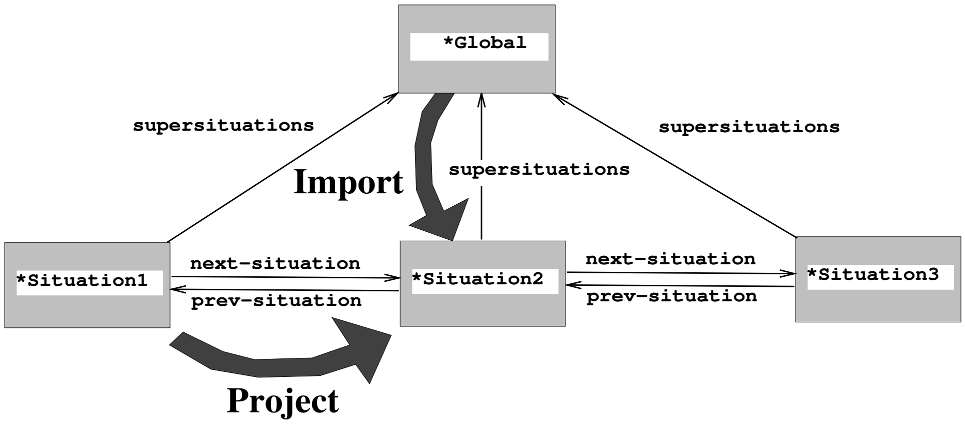



Figure 1: Situations and their relationships in KM.

can be reasoned about. The global KB is also considered a situation, with name `*Global`, and is the supersituation of all other situations. These relationships are shown in Figure 1, along with a glimpse of the `next-situation` and `prev-situation` relationships and the projection operation. We will ignore projection for now, and introduce later in this manual.

When in a (non-global) situation, KM prints the situation name as a prefix to the KM prompt as shown below. We will sometimes refer to assertions made in a (non-global) situation as "local" assertions, and assertions in the global situation as "global" assertions.

At the KM prompt, the user can *create* a situation, and *enter* a situation using the command `(in-situation` *situation*`)`:

;;; "Joe is a person, born in 1963."
;;; $isa(*Joe, Person) \wedge birthdate(*Joe, 1963)$
```
KM> (*Joe has                                   ; Global KB assertion
        (instance-of (Person))                  ; (visible to all Situations)
        (birthdate (1963)))
```

;;; "Create a situation."
;;; $\exists s\ isa(s, Situation)$
```
KM> (a Situation)
(_Situation5)
```

;;; "Enter that situation."
```
KM> (in-situation _Situation5)
(Changing to situation _Situation5)
```

;;; "What is Joe's birthdate (in this situation)?"
;;; $\{\ d \mid holds\text{-}in(birthdate(*Joe, d), \_Situation5)\ \}$
```
[_Situation5] KM> (the birthdate of *Joe)       ; Global facts are visible
(1963)                                          ; from a local situation
```

;;; "Joe is happy (in this situation)."
;;; $holds\text{-}in(mood(*Joe, *Happy), \_Situation5)$
```
[_Situation5] KM> (*Joe has (mood (*Happy)))    ; Make a local assertion
```

;;; "What is Joe's mood (in this situation)?"
;;; $\{\ m \mid holds\text{-}in(mood(*Joe, m), \_Situation5)\ \}$

```
[_Situation5] KM> (the mood of *Joe)
(*Happy)                                        ; Assertion visible here

;;; "Return to the global situation."
[_Situation5] KM> (in-situation *Global)
(Changing to situation *Global)

;;; "What is Joe's (permanent) mood?"
;;; { m | mood(*Joe, m) }
KM> (the mood of *Joe)                          ; Local assertions are not
NIL                                             ; visible to the global KB

;;; "Create and enter a new situation."
KM> (in-situation (a Situation))
(Changing to situation _Situation6)

;;; "Joe is sad (in this new situation)."
;;; holds-in(mood(*Joe, *Sad), _Situation6)
[_Situation6] KM> (*Joe has (mood (*Sad)))      ; Alternative assertion

;;; "What is Joe's mood (in this situation)?"
;;; { m | holds-in(mood(*Joe, m), _Situation6) }
[_Situation6] KM> (the mood of *Joe)
(*Sad)

;;; "Return to the global situation."
[_Situation6] KM> (in-situation *Global)
(Changing to situation *Global)

KM>
```

For convenience, the commmands `(new-situation)` both creates and enters a new situation; `(curr-situation)` returns the name of the current situation, and `(global-situation)` returns to the global KB:

```
KM> (new-situation)
(Changing to situation _Situation7)

[_Situation7] KM> (curr-situation)
(_Situation7)

[_Situation7] KM> (global-situation)
(Changing to situation *Global)

KM>
```

## 2.2 Viewing the Contents of Situations

As for non-situational KM, the contents of frames can be viewed with the `(showme` *expr*`)` command. When situations are used, however, this command also prints out the situation-specific assertions, wrapped in an `(in-situation ...)` expression:

```
KM> (showme *Joe)
(*Joe has
```

```
  (instance-of (Person))
  (birthdate (1963)))

(in-situation _Situation6
  (*Joe has
    (mood (*Sad))))

(in-situation _Situation5
  (*Joe has
    (birthdate (1963))
    (mood (*Happy))))
```

Note that, as for non-situational KM, KM only shows the facts explicitly asserted or computed for this frame in response to queries, not all the deductively implied facts.
The command `(showme-here` *expr*`)` just shows the current situation's part of *expr*:

```
KM> (in-situation _Situation5)

[_Situation5]> (showme-here *Joe)
(in-situation _Situation5
  (*Joe has
    (birthdate (1963))
    (mood (*Happy))))
```

In addition, as for non-situational KM, `(save-kb "myfile.km")` writes out the current state of the KB to the file "myfile.km", and `(write-kb)` writes it out to the standard output. `save-kb` is often useful for debugging, as standard editor functions can then be used to search the saved file when viewed in a text editor.

## 2.3 Quantifying over Situations

In addition to explicitly entering a situation to ask situation-specific queries, the user can issue situation-specific queries by passing the query as a second argument to the `(in-situation ...)` command. In this case, KM will temporarily enter the situation to evaluate the query, then return to the global situation:

;;; "What is Joe's mood in _Situation5?"
;;; $\{ m \mid \textit{holds-in}(\textit{mood}(*Joe, m), \_Situation5) \}$

```
KM> (in-situation _Situation5 (the mood of *Joe))
(*Happy)
```

;;; "What is Joe's mood in _Situation6?"
;;; $\{ m \mid \textit{holds-in}(\textit{mood}(*Joe, m), \_Situation6) \}$

```
KM> (in-situation _Situation6 (the mood of *Joe))
(*Sad)
```

Note, thus, that KM can maintain knowledge of multiple situations at the same time.

We can use the `(in-situation ...)` expression to quantify over situations, for example to ask "Is it ever the case that...?" questions. For instance:

;;; "Is Joe ever happy?"
;;; (expressed as: "Does there exist a situation in which Joe is happy?")
;;; $oneof(\{ s \mid isa(s, Situation) \wedge \textit{holds-in}(\textit{mood}(*Joe, *Happy), s) \})$[1]

[1] where $oneof()$ is a deterministic but unspecified function mapping a set onto one of its members (see the User Manual for further details).

```
KM> (oneof (the all-instances of Situation)
         where (in-situation It ((the mood of *Joe) = *Happy))))
(_Situation5)                              ; such a Situation exists (ie. "Yes")
```

Here, `(the all-instances of Situation)` computes the situations to quantify over (`all-instances` is a built-in slot returning all the instances of a class, see the User Manual). We later show additional ways that a set of situations can be created and referred to automatically. This ability to manipulate situations as KB objects in their own right has many useful applications, as we will illustrate later in this manual.

# 3 The Situation Hierarchy

## 3.1 The Semantics of Situations

Semantically, making an assertion $p$ in a situation $s$ is to assert the proposition $\textit{holds-in}(p, s)$ in a single theory corresponding the entire knowledge-base. This includes assertions in the 'global KB', which is treated as a special situation with name `*Global`. Thus, strictly, asserting $p$ in the global situation (i.e., at the normal `KM>` prompt) asserts the proposition $\textit{holds-in}(p, *Global)$, although we write just $p$ as a shorthand for this in the KM manuals.

Situations are themselves instances, and can be organized into a situation hierarchy using the built-in slot `supersituations` (and its inverse `subsituations`), which relates a situation to its parent situations. Note that this is not the same as the `superclasses` relation (`superclasses` relates classes to classes, while `supersituations` relates situation instances to situation instances).

By default, new situations are subsituations of the `*Global` situation. A situation can 'see' the assertions in all its supersituations, i.e., subsituations acquire all the assertions in their supersituations. The semantics of this is described by the axiom schema (i.e., by the infinite set of axioms which the schema specifies):

"If a proposition *prop* holds in situation $s$, then it also holds in each subsituation $s'$"
$\forall s, s'\ \textit{holds-in}(\textit{prop}, s) \land \textit{subsituations}(s, s') \rightarrow \textit{holds-in}(\textit{prop}, s')$

for all possible first-order logic statements *prop*.

Situations can have multiple supersituations, i.e., the situation 'hierarchy' can be a lattice. All propositions from all supersituations hold in a situation.

## 3.2 Inference

When finding a frame's slot-value in a situation $S$, KM also looks in that situation's supersituation(s) for information. This includes both 'own properties' (*frame* `has ...`) *and* 'member properties' (`every` *frame* `has ...`) for that frame. Thus, an instance's slot can acquire information from a supersituation in two ways: first, it will acquire the values on that instance's slot in the supersituation; second, its *classes* will acquire the expressions (rules) on those classes in the supersituation, which will subsequently be inherited down to that instance and evaluated. Thus a situation acquires all the *axioms* of its supersituations (as we specified in Section 3.1), not just ground literals. This is important because the same axiom may imply different things in different situations, e.g., $(x \leftrightarrow y)$ implies $y$ in situations where $x$ is true, and $\neg y$ in situations where $x$ is false. It also contrasts with projection, described later, where only ground literals are projected between situations.

KM's tracing mechanism allows the user to observe how supersituations are used when compute the answer to a question. An example is illustrated below.

;;; "A person's age (in years) is today's year minus their year of birth."
;;; $\forall p, y, y', a \; isa(p, Person) \wedge year(*Todays\text{-}Date, y) \wedge year\text{-}of\text{-}birth(p, y') \wedge a = y - y'$
;;; $\rightarrow age(p, a)$

```
KM> (every Person has
      (year-of-birth ((a Number)))                              ; (in years)
      (age (((the year of *Todays-Date) - (the year-of-birth of Self)))))
```

;;; "Fred was born in 1963."
;;; $isa(*Fred, Person) \wedge year\text{-}of\text{-}birth(*Fred, 1963)$

```
KM> (*Fred has
      (instance-of (Person))
      (year-of-birth (1963)))
```

;;; "Create and enter a new situation."

```
KM> (new-situation)
```

;;; "In this situation, the year is 2000."
;;; $holds\text{-}in(year(*Todays\text{-}Date, 2000), \_Situation0)$

```
[_Situation0] KM> (*Todays-Date has
                         (year (2000)))
```

;;; "Switch on tracing in KM."

```
[_Situation0] KM> (trace)
```

Now in the next query, note that Fred acquires year-of-birth 1963 from the supersituation. Note also that Fred also inherits year-of-birth `(a Number)` from the class Person, i.e. both the "rule" on Person and the "value" on Fred are passed down from the global situation to the local situation, and combined.

;;; "What year was Fred born?"

```
[_Situation0] KM> (the year-of-birth of *Fred)
1 -> (the year-of-birth of *Fred)
1  (1) Look in supersituation(s)
2  -> (in-situation *Global (the year-of-birth of *Fred))
2  <- (1963)                  "(in-situation *Global (the year-of-birth of *Fred))"
1  (2) From inheritance: (a Number)
1  (1-2) Combine 1-2 together
2  -> ((1963) && ((a Number)))
2  <- (1963)                  "((1963) && ((a Number)))"
1 <- (1963)                   "(the year-of-birth of *Fred)"
(1963)
```

;;; "How old is Fred in this situation?"
;;; Again note the multiple sources of information for computing Fred's year-of-birth, and age.

```
[_Situation0] KM> (the age of *Fred)
1 -> (the age of *Fred)
1  (1) Look in supersituation(s)
2  -> (in-situation *Global (the age of *Fred))
2  <- FAIL!                   "(in-situation *Global (the age of *Fred))"
1  (2) From inheritance: ((the year of *Todays-Date) - (the year-of-birth of *Fred))
2  -> ((the year of *Todays-Date) - (the year-of-birth of *Fred))
3   -> (the year of *Todays-Date)
3    (1) Look in supersituation(s)
```

```
4    -> (in-situation *Global (the year of *Todays-Date))
4    <- FAIL!                  "(in-situation *Global (the year of *Todays-Date))"
3    (2) Local value(s): 2000
3   <- (2000)                  "(the year of *Todays-Date)"
3   -> ((the year-of-birth of *Fred))
4    -> (the year-of-birth of *Fred)
4     (1) Look in supersituation(s)
5     -> (in-situation *Global (the year-of-birth of *Fred))
5     <- (1963)                "(in-situation *Global (the year-of-birth of *Fred))"
4     (2) Local value(s): 1963
4     (3) From inheritance: (a Number)
4     (1-3) Combine 1-3 together
5     -> ((1963) && ((a Number)))
5     <- (1963)                "((1963) && ((a Number)))"
4    <- (1963)                 "(the year-of-birth of *Fred)"
3   <- (1963)                  "((the year-of-birth of *Fred))"
2  <- (37)                     "((the year of *Todays-Date) - (the year-of-birth of *Fred))"
1 <- (37)                      "(the age of *Fred)"
(37)
```

In the above, KM is searching in both the local and global situations for all the information it needs (hence the long trace). It finds today's date in the local situation, and then three values for Fred's year of birth, namely in the global situation (`1963`), the local situation (`1963`), and by inheritance from `Person` (`(a Number)`), which it then combines. Although it might seem redundant here to find the same value in three different ways, in general these three sources might provide different values (which would then combine to produce a multi-valued answer), hence this search is necessary in general. In this particular case, however, *we* know that a person's year of birth is not going to vary between situations (i.e., it is not a fluent), and we would like to tell KM this to save it some work. We explain how to do this shortly in the next subsection.

Finally KM makes the necessary subtraction to find Fred's age, returning `37`.

By default, KM does not show the details of the computations in other situations besides the current situation. For example, above, details of the computation of `(in-situation *Global (the year-of-birth of *Fred))` are not shown, although this itself might have involved several inference steps in the global situation. To view such details, toggle the trace mode using the tracing option `+S`.

### 3.3 Fluents and Non-Fluents

Slots whose values are situation-dependent are called **fluents** (for example, `age` above). In addition, as we discuss later, there is a special type of fluent which we call an **inertial fluent**, namely a slot whose values are situation-dependent *and* whose values persist from one situation to another, unless there is evidence to the contrary. For example we might decide to treat `age` as an inertial fluent, if we want KM to assume someone's age (in years) in a situation is the same as in the immediately preceding situation, unless there is evidence to the contrary. However, `best-move` for a chess game might be modeled as a non-inertial fluent, as the best chess move in any given situation depends solely on the current state of the chess board, and has nothing to do with earlier best moves. Finally, `year-of-birth` is an example of a non-fluent, as its value is the same in all situations. We discuss inertial fluents and situation sequences in more detail in Section 6.

The user can declare the fluent status of a slot using the built-in slot's slot `fluent-status`, which can have one of three values:

| <u>Slot's `fluent-status`</u> | <u>Meaning</u> |
|---|---|
| `*Non-Fluent` | slot value(s) are the same in all situations |
| `*Fluent` | slot value(s) may vary between situations |
| `*Inertial-Fluent` | slot value(s) may vary between situations, but should persist from one situation to the temporally next situation unless there is evidence to the contrary. |

For example:

```
;;; "year-of-birth is not a fluent."
KM> (year-of-birth has
       (instance-of (Slot))
       (fluent-status (*Non-Fluent)))
```

By default, KM considers *all* slots to be inertial fluents (i.e. `fluent-status` is `*Inertial-Fluent`). The user can change the fluent status for specific slots using `fluent-status` declarations (as above), or, if he/she prefers, change the default for *all* slot using the function `(default-fluent-status` *new-status*`)`, and then declare exceptions to this new status instead. For some applications, this latter approach may be more concise. `(default-fluent-status` *new-status*`)` is used as follows:

```
;;; "By default, consider all slots to be non-fluents."
KM> (default-fluent-status *Non-Fluent)
By default, slots are now considered to be non-fluents.
```

Declaring which slots are/are not fluents requires a bit of extra work by the knowledge engineer, but allows KM to avoid redundant computations: KM will only repeat computations in supersituations for fluents (both inertial and non-inertial), and only repeat computations in previous situations for inertial fluents. This makes KM's reasoning faster, and also simplifies the trace for the user when debugging.

### 3.4 Situation-Specific Slots

Similarly, there are some slots which have *no* meaningful value in the global situation, for instance `age` in the above example. Nevertheless, KM by default does not realize this, and will thus needlessly try (and fail) to compute such slots' values in the global situation when queried about it. This behavior can be suppressed by declaring the slot `situation-specific`, meaning that it has no meaningful values in the global situation, i.e., $\forall x, y\ \neg slot(x, y)$:

```
KM> (age has
      (instance-of (Slot))
      (situation-specific (t)))
```

This will make reasoning a tiny bit more efficient, and also simplify tracing of KM's inference. Queries about situation-specific slots in the global situation will automatically return `NIL`.

There is also a second usage of the `situation-specific` attribute, which is as a 'cheap trick' to avoid problems with negation-as-failure and the closed-world assumption. Consider the rule:

```
(every Person has
  (is-adult ((if ((the age of Self) >= 18) then *Yes else *No))))
```

Now in the global situation, a person does not have a value for age (as it varies from non-global situation to non-global situation). Thus a query for whether a person is an adult will necessarily return the answer *No (as the test fails), which is then passed down to all non-global situations! This is not what we intended, as a situation may exist where the person's age is greater than 18, and thus a contradiction will arise. The best way to avoid this is to reformulate the rule so it does not rely on negation as failure:

```
(every Person has
  (is-adult ((if ((the age of Self) >= 18) then *Yes
             else (if ((the age of Self) < 18) then *No)))))
```

However, another alternative, if we are sure that a person's age *will* be computable in all non-global situations, is to prevent KM from computing values for this slot in the global situation. This can be done again by setting the slot's `situation-specific` property to `t`:

```
KM> (is-adult has
      (instance-of (Slot))
      (situation-specific (t)))
```

In general, though, the first approach of reformulating the rule is the preferred way of dealing with this issue.

# 4 More About Situation Instances and Classes

This Section describes more advanced features for declaring situations, and can be skipped if only an introductory level to Situations is needed.

## 4.1 Situation Instances

Situations are instances (of the class `Situation`), and behave just like any other instances in KM. This means that they can be given explicit names, can be qualified using a `(`*situation* `has...)` or `(a Situation with...)` expression, and can be organized into classes (subclasses of the class `Situation`).

It is important to be aware of which assertions are being made in the global situation, and which in a local situation, especially when making assertions *about* situations themselves. This is particularly important because, as mentioned earlier, assertions made in the global situation are universal, and will be imported into all other situations. We elaborate on this further here with some examples.

We can attach slot-values to a situation, for example:

;;; "Create a situation whose time/date is the morning of 4-20-00."
;;; $\exists s\ isa(s, Situation) \wedge date(s, \text{4-20-00}) \wedge time(s, *Morning)$

```
KM> (a Situation with
      (date (4-20-00))
      (time (*Morning)))
(_Situation2)
```

We can also create situations with explicit names, e.g.:

;;; "Pete's Thursday Morning is a situation whose time/date is the morning of 4-20-00."
;;; $isa(*Petes\text{-}Thursday\text{-}Morning, Situation) \wedge date(*Petes\text{-}Thursday\text{-}Morning, \text{4-20-00}) \wedge$
;;; $time(*Petes\text{-}Thursday\text{-}Morning, *Morning)$

```
KM> (*Petes-Thursday-Morning has
        (instance-of (Situation))
        (date (4-20-00))
        (time (*Morning)))
```

;;; "Enter this situation."

```
KM> (in-situation *Petes-Thursday-Morning)
```

;;; "Pete was located on a chair in front of a computer on Thursday Morning."
;;; $\exists c, d$ $holds\text{-}in($ $isa(c, Chair) \wedge isa(d, Computer) \wedge location(*Pete, c) \wedge in\text{-}front\text{-}of(c, d),$
;;; $*Petes\text{-}Thursday\text{-}Morning$ $)$

```
[*Petes-Thursday-Morning] KM> (*Pete has
                                (location ((a Chair with
                                             (in-front-of ((a Computer)))))))
```

Now, the situation instance itself can refer to objects *within* the situation partition of the KB. For example, I could say the agent in this situation is Pete by asserting:

```
[*Petes-Thursday-Morning] KM> (global-situation)
```

;;; "Pete is the main participant in the *Petes-Thursday-Morning situation."
;;; $main\text{-}participant(*Petes\text{-}Thursday\text{-}Morning, *Pete)$

```
KM> (*Petes-Thursday-Morning has
       (main-participant (*Pete)))
```

This declares `*Pete` as the main participant in `*Petes-Thursday-Morning`. Note that this is a statement *about* a situation instance, rather than an assertion *within* the situation partition which that instance refers to. It can be thought of as meta-knowledge about a situation, residing in the global KB.

Similarly, statements *within* a situation partition can refer to objects *associated* with the situation instance that denotes it, for example the second assertion below:

;;; "Create a situation whose main participant is a person."
;;; $\exists s, p$ $isa(s, Situation) \wedge isa(p, Person) \wedge main\text{-}participant(s, p)$

```
KM> (a Situation with
      (main-participant ((a Person))))
(_Situation1)
```

;;; "In that situation, the main participant is located on a chair."
;;; $\forall p$ $main\text{-}participant(\_Situation1, p) \rightarrow \exists c$ $holds\text{-}in($ $isa(c, Chair) \wedge location(p, c), \_Situation1)$

```
KM> (in-situation _Situation1
        ((the main-participant of _Situation1) has (location ((a Chair)))))
```

Here, the assertion *within* the situation partition refers to the main participant of the situation instance denoting it.

KM allows these two kinds of assertions (*about* and *within* a situation) to be combined into a single structure, by placing the `in-situation` assertions on the special slot called `assertions` on situation instances. Assertions should be quoted KM expressions, and are a list of statements which hold *within* the situation which the instance denotes. For example, the above two statements could be combined into a single statement:

;;; "Create a situation whose main participant is a person who, in that situation,
;;; is located on a chair."

;;; $\exists s, p\ isa(s, Situation) \wedge isa(p, Person) \wedge main\text{-}participant(s, p) \wedge$
;;; $(\ \forall p\ main\text{-}participant(s, p) \rightarrow \exists c\ holds\text{-}in(\ isa(c, Chair) \wedge location(p, c), s)\ )^2$

```
KM> (a Situation with
        (main-participant ((a Person)))
        (assertions ('((the main-participant of Self) has (location ((a Chair)))))))
(_Situation0)
```

The above statement creates `_Situation0`, and asserts that the main participant is on a chair within it. The following query confirms that this assertion has been made:

;;; "Where is the main participant located in _Situation0?"
;;; $\{\ l \mid \exists p\ holds\text{-}in(\ main\text{-}participant(\_Situation0, p) \wedge location(p, l), \_Situation0)\ \}$

```
KM> (in-situation _Situation0 (the location of (the main-participant of _Situation0)))
(_Chair8)
```

Note that the alternative below would not capture what we want, althought it looks similar on first glance:

;;; "Create a situation whose main participant is a person (permanently) located on a chair."
;;; $\exists s, p, c\ isa(s, Situation) \wedge isa(p, Person) \wedge isa(c, Chair) \wedge main\text{-}participant(s, p) \wedge location(p, c)$

```
KM> (a Situation with
      (main-participant ((a Person with (location ((a Chair)))))))
```

This would be problematic because it asserts *in the global situation* that the person is on the chair, i.e. the person is permanently on the chair, rather than making the assertion in the local situation. This is not what we intended to say. Care is needed!

## 4.2 Situation Classes

The user can also define *classes* of situations. Just as (`in-situation` *situation expr*) evaluates *expr* in *situation*, (`in-every-situation` *situation-class expr*) will evaluate *expr* for every instance of *situation-class* created. This evaluation is done once only for each situation, at the time the situation is created. Just as `Self` refers to the instance under consideration in an `every` clause, so the keyword `TheSituation` refers to the situation instance under consideration in an `in-every-situation` clause, as illustrated below.

In the below example, we represent a specific, ongoing action (here, "falling") as a situation, and also define the class of all such situations (i.e. all "falling" situations). This is one way of describing ongoing activities in KM.

;;; "Falling situations are situations."
;;; $superclasses(Falling\text{-}Situation, Situation)$

```
KM> (Falling-Situation has (superclasses (Situation)))
```

;;; "Every falling situation involves a person."[3]
;;; $\forall s\ isa(s, Falling\text{-}Situation) \rightarrow \exists p\ isa(p, Person) \wedge agent(s, p)$

```
KM> (every Falling-Situation has
      (agent ((a Person))))
```

;;; "In falling situations, the falling person feels scared."
;;; $\forall s\ isa(s, Falling\text{-}Situation) \rightarrow holds\text{-}in(\ \forall p\ agent(p, s) \rightarrow feelings(p, *Scared), s)$

[2] This shows the built-in semantics of $assertions(s, p)$, namely the quoted assertion $p$ holds in situation $s$.
[3] i.e. we are really describing `People-Falling-Situation` rather than `Falling-Situation`

```
KM> (in-every-situation Falling-Situation
        ((the agent of TheSituation) has (feelings (*Scared))))
```

This last statement is equivalent to (and is in fact internally translated to):

```
;;; (Alternative syntax, but same semantics as the previous statement)
KM> (every Falling-Situation has
       (assertions ('((the agent of Self) has (feelings (*Scared))))))
```

On creation of a situation instance of this class, the situation-specific assertions are made into that situation's partition, as shown below:

```
;;; "Pete is a person."
;;; isa(*Pete, Person)
KM> (*Pete has (instance-of (Person)))

;;; "There is a situation in which Pete is falling."
;;; ∃s isa(s, Falling-Situation) ∧ agent(s, *Pete)
KM> (a Falling-Situation with (agent (*Pete)))
(COMMENT: Evaluating (in-situation _Falling-Situation9
             ((the agent of _Falling-Situation9) has (feelings (*Scared)))))
(_Falling-Situation9)

;;; "What are Pete's feelings in that situation?"
;;; { f | holds-in(feelings(*Pete, f), _Falling-Situation9) }
KM> (in-situation _Falling-Situation9 (the feelings of *Pete))
(*Scared)
```

As before, note the below does not capture what we intended to say:

```
;;; "All falling situations involve a (permanently) scared person."
;;; ∀s isa(s, Falling-Situation) → ∃p isa(p, Person) ∧ agent(s, p) ∧ feelings(p, *Scared)
KM> (every Falling-Situation has
      (agent ((a Person with (feelings (*Scared))))))
```

because it asserts in the *global* situation that the person is scared, i.e. the person is permanently scared, whereas our intent was to say the person is scared just *during* the falling (i.e. is a situation-specific assertion).

# 5 Representing Actions

## 5.1 Representing Actions

### 5.1.1 Preconditions, Add and Delete Lists

Actions are events which change the state of the world. Thus, the application of an action in a situation is modeled by the creation of a new situation, reflecting the new world state after the action has been performed. Note that in this approach, the situation in which the action itself is happening is not modeled (one might think of the actions as being instantaneous). Later, in Section 11.5, we will suggest a variation of this approach in which actions-in-progress are also modeled (in fact, where we remove the dichotomy between actions and states of affairs), but for now we continue modeling the world as comprising of situations, and actions which change situations.

In KM, actions are described using four lists, namely the 'pcs-list' (preconditions list), the 'ncs-list' (negated preconditions list), the 'add-list' (add list), and the 'del-list' (delete list). The pcs-list contains a list of ground literals which are necessarily true before an action is performed, the ncs-list contains a list of propositions which are necessarily false, the add-list contains propositions which are necessarily true after the action is performed (i.e., "become true" as a result of the action), and the del-list contains those which are false (i.e., "become false"). (We describe this formally in Section 5.3). These lists are stored on the `pcs-list`, `ncs-list`, `add-list`, and `del-list` slots respectively on the frame representing the action. A proposition is a reified expression (ie. an expression represented as an object), and allows us to make statements *about* that proposition P, for example "Fred believes P" – or, for our purposes here, "The result of doing X is P".

### 5.1.2 Representing Propositions

A proposition is represented in KM by the structure `(:triple` *frame slot value*`)`, which denotes (but does not make) the assertion that `frame`'s `slot` includes `value`. For example:

```
;;; The proposition "Pete is happy"
;;; "state(*Pete,*Happy)", where the string is just a constant denoting the proposition.
KM> (:triple *Pete state *Happy)
((:triple *Pete state *Happy))
```

Note that the above does not assert that "Pete is happy" is true; it merely denotes the statement itself (which may be true or false).

When processing a `:triple` expression, KM *does* evaluate the three components (frame, slot, value) of the triple. For example, `(:triple Self state *Happy)` does not denote the proposition `"state(Self,*Happy)"`, but the proposition `"state(<p>,*Happy)"`, where `p` is the instance which `Self` denotes. In our logic notation notation in this manual, `<>` signifies this "unquoting", i.e., we write `"<p>"` to denote the value of $p$, rather than the string `"p"`. Some examples are shown below:

```
;;; "Every person believes that he/she is happy."
;;; ∀p isa(p, Person) → belief(p,"state(<p>,*Happy)")
KM> (every Person has
      (belief ((:triple Self state *Happy))))

;;; "Every person believes that all his/her pets are happy."
;;; ∀p isa(p, Person) →
;;;       ∀t has-pets(p, t) → belief(p,"state(<t>,*Happy)")
KM> (every Person has
      (belief ((forall (the has-pets of Self)
                       (:triple It state *Happy)))))

;;; "Every person believes his/her house is beautiful."
;;; ∀p, h isa(p, Person) ∧ house-lived-in(p, h) → belief(p,"appearance(<h>,*Beautiful)")[4]
KM> (every Person has
      (belief ((:triple (the house-lived-in of Self) appearance *Beautiful))))
```

`(assert (:triple f s v))` is equivalent to the assertion `(f has (s (v)))`.

[4] This KM expression assumes that a person lives in at most one house.

### 5.1.3 Manipulating Propositions

To assert that a triple is actually true, the KM function `(assert` *expr*`)` is used, where *expr* evaluates to zero or more triples. To test the truth-value of a triple, the KM functions `(is-true` *expr*`)`, `(all-true` *expr*`)`, and `(some-true` *expr*`)` should be used, where *expr* evaluates to zero or more triples. `all-true` tests whether all those triples are true, `some-true` tests that at least one is true, and `is-true` is equivalent to `all-true`, except expects *expr* to evaluate to exactly one triple and will generate an error otherwise. These tests denote success by returning the constant `t`, and return `NIL` otherwise. For example:

```
;;; "Is the proposition that 'Pete is happy' true?"
KM> (is-true (:triple *Pete state *Happy))
NIL                                                      ; No

;;; "Pete is happy"
KM> (assert (:triple *Pete state *Happy))
(:triple *Pete state *Happy)

;;; "Is the proposition that 'Pete is happy' true?"
KM> (is-true (:triple *Pete state *Happy))
(t)                                                      ; Now, yes!

;;; Another way to do the same test:
KM> ((the state of *Pete) = *Happy)
(t)
```

### 5.1.4 Representing the Effects of Actions using Propositions

To describe the preconditions and effects of actions, propositions are placed on the `pcs-list`, `ncs-list`, `add-list`, and `del-list` slots of those actions. As an example, the action of switching on a switch might be represented:

```
;;; "Switching on is a type of action."
;;; superclasses(Switching-On, Action)
(Switching-On has (superclasses (Action)))

;;; "The effect of a Switching-On is that the switch's position becomes Up."
;;; ∀s, o isa(s, Switching-On) ∧ object(s, o) →
;;;         pcs-list(s,"position(<o>,*Down)") ∧
;;;         del-list(s,"position(<o>,*Down)") ∧
;;;         add-list(s,"position(<o>,*Up)")
(every Switching-On has
  (object ((a Switch)))
  (pcs-list ((:triple (the object of Self) position *Down)))
  (del-list ((:triple (the object of Self) position *Down)))
  (add-list ((:triple (the object of Self) position *Up))))
```

### 5.1.5 The Situation-Specificity of Actions

It is important to note that the effects of an action instance may vary, depending on the situation in which it is performed. For example, the effect of toggling a switch will vary, depending on the current position of a switch. As a result, it does not always make sense to ask what an action's effects are in the global situation, and so KM will ignore queries for the action's pcs/ncs/add/del lists if issued in the global situation. This ignoring happens because these four slots are internally declared as 'situation-specific', meaning they only have meaningful values in non-global situations (see Section 3.4). It is done to protect the user from possible problems with negation as failure, which might arise if KM computed these lists in the global situation where the situation-specific facts were absent.

In addition, these four slots are *non-inertial* fluents (i.e. KM does not project their values from one situation to another), to ensure KM does not attempt to compute and combine the effects of a single action instance in situations other than the one in which it is to be performed.

### 5.1.6 Ramifications

In general, an action may also have ramifications, i.e., 'indirect' effects as well as the 'direct' effects described by the pcs/ncs/add/del lists. Ramifications are sometimes called "state constraints" also in the literature. For example, switching a switch on may have the direct effect of the switch's position now being up, and the indirect effect (ramification) that a light comes on. What is a direct effect and what is a ramification is a modeling decision, rather than an intrinsic property of actions, determined by which effects the knowledge engineer explicitly listed on the action, and which he/she left to be deduced from those explicit effects. This makes representing actions much easier. In general, an action may have many indirect effects – however, the `add-list` and `del-list` only need to describe *direct* effects. Ramifications, if there are any, will be recomputed by KM in the new situation from those direct effects (plus other persistent facts from the old situation, discussed later in Section 6). Note that the semantics of actions with ramifications is more complicated, as sometimes the resulting situation can have multiple, possible extensions (see Section 11.3). Care is needed!

## 5.2 Relating Situations and Actions

In Situation Calculus, the situation resulting from doing action $a$ in situation $s$ is denoted by the function $do(a, s)$. In KM, we express this relationship differently using the predicate *next-situation*$(s, s', a)$, where $s'$ is the result of doing $a$ in $s$ (i.e., $s' = do(a, s)$). As this is a non-binary predicate, we use KM's `:args` notation to denote this (see the KM User Manual). The inverse slot of `next-situation` is `prev-situation`, and the "second inverse" (`inverse2`) for this ternary predicate is `before-situation`, itself with inverse `after-situation`. The semantics of this are below, and illustrated in Figure 2:

$$\forall s, s', a\ \textit{next-situation}(s, s', a) \leftrightarrow \textit{prev-situation}(s', s, a)$$
$$\leftrightarrow \textit{before-situation}(a, s, s') \leftrightarrow \textit{after-situation}(a, s', s)$$

These relationships are built-in, and the assertions are made automatically by KM after an action is "performed" using KM's `do` command (Section 5.3.2). If they were to be made manually, however, they would look:

```
;;; "_Situation2 is the result of doing _Action1 in _Situation1."
;;; next-situation(_Situation1, _Situation2, _Action1)
KM> (_Situation1 has
      (next-situation ((:args _Situation2 _Action1))))

;;; "What is the next situation (and the action which led to it) of Situation1?"
KM> (the next-situation of _Situation1)
((:args _Situation2 _Action1))
```

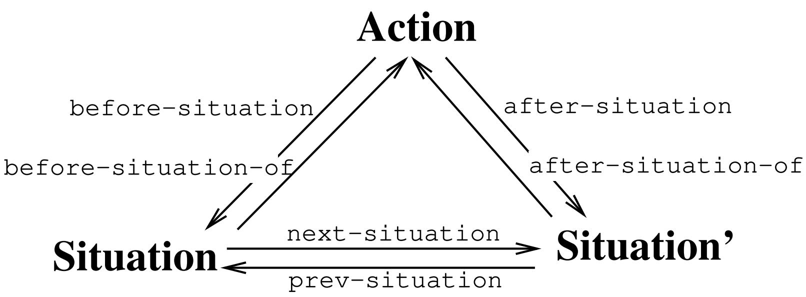



Figure 2: The relationship between situations and actions.

```
;;; "What situation was immediately before Action1 was performed?"
KM> (the before-situation of _Action1)
((:args _Situation1 _Situation2))          ; i.e. ((:args <before-sitn> <after-stn>))
```

As described in the KM User Manual, there are some additional forms which can be used for manipulating ternary relations and `:args` structures. To recap these, `(the1` *slot* `of...)` retrieves the second argument of a ternary relation (*slot*) given the first, `(the2` *slot* `of...)` retrieves the third argument given the first, and `(the` *slot* `of...)` retrieves both the second and third arguments given the first, bundled into a `(:args ...)` structure. Similarly, `(the1 of (:args ...))` extracts the first element in an `:args` structure, and `(the2 of (:args ...))` extracts the second. Use of these is illustrated below:

```
;;; "What is the next situation of Situation1?"
KM> (the1 next-situation of _Situation1)
(_Situation2)

;;; "What action was performed in Situation1?"
KM> (the2 next-situation of _Situation1)
(_Action1)

;;; "Which situation followed from performing Action1 in Situation1?"
KM> (the1 of                                          ; Take first element of ...
       (theoneof (the next-situation of _Situation1) ; the next-situation structure..
           where ((the2 of It) = _Action1)))        ; whose second element             is _Action1
(_Situation2)
```

In the above example, we have just considered there to be a single next situation, resulting from performing a single action `_Action1` in `_Situation1`. However, KM will also allow multiple, alternative next situations, resulting from performing different, alternative actions, thus allowing alternative futures to be explored. We describe how this can be done later in Section 9.

## 5.3 The Semantics of Actions

### 5.3.1 Declarative Semantics

If $s'$ is the situation resulting from doing action $a$ in situation $s$ (i.e., $\textit{next-situation}(s, s', a)$ is true), then we can describe the semantics of actions as follows, quantifying over ground propositions $p$:

"All the preconditions will hold in situation s."
$\forall s, s', p, a \ \textit{holds-in}(\textit{pcs-list}(a, p), s) \wedge \textit{next-situation}(s, s', a) \rightarrow \textit{holds-in}(p, s)$

"All the negated preconditions will be false in situation s."
$\forall s, s', p, a \ \textit{holds-in}(\textit{ncs-list}(a, p), s) \wedge \textit{next-situation}(s, s', a) \rightarrow \textit{holds-in}(\neg p, s)$

"All the add effects will be true in situation s'."
$\forall s, s', p, a \ \textit{holds-in}(\textit{add-list}(a, p), s) \wedge \textit{next-situation}(s, s', a) \rightarrow \textit{holds-in}(p, s')$

"All the delete effects will be false in situation s'."
$\forall s, s', p, a \ \textit{holds-in}(\textit{del-list}(a, p), s) \wedge \textit{next-situation}(s, s', a) \rightarrow \textit{holds-in}(\neg p, s')$

#### 5.3.2 Procedural Implementation

```
(do expr)
(do-and-next expr)
```

Procedurally, the user directs KM to compute these effects by executing the built-in command `(do` *expr*`)` from within a situation (excluding the global situation), where *expr* is a KM expression evaluating to an action instance. This causes KM to build a new situation reflecting the result of doing this action, and can be loosely thought of as 'executing' the action. More specifically, `(do` *expr*`)` causes KM to:

1. create a new situation, which is the `next-situation` of the current situation.
2. 
   - assert the preconditions in the action's `pcs-list` in the current situation,
   - assert constraints that the negated preconditions in the action's `ncs-list` are not true in the current situation (using `(<>` *expr*`)` constraint expressions, described in the User Manual),
   - 'retract'[5] the propositions in the action's `del-list` in the new situation, and
   - asserts the propositions in the action's `add-list` in the new situation.

Note that KM asserts, rather than checks the truth of, the action preconditions, i.e., preconditions are treated as assertions, not tests. Section 7 describes how preconditions can also be used as tests, to see if an action is possible. This, of course, is only appropriate in the special case when the KB has complete knowledge about action preconditions, i.e., when if a precondition is nprovable, it can be assumed false.

The KM command `(do-and-next` *expr*`)` is similar to `(do` *expr*`)`, except it additionally causes KM to change to the new situation created.

Performing the null action `(do-and-next nil)` creates and changes to a new situation with no changes, but still recorded as the `next-situation` of the original situation. For legibility, the command `(next-situation)` simply does (i.e., is equivalent to) `(do-and-next nil)`.

[5]Implemented by asserting the constraint that these facts do not hold the new situation, using `(<>` *expr*`)` constraint expressions. Note that KM does not literally delete (retract) these propositions, but instead these constraints block their projection from the old to the new situation (thus appearing to delete them). This implementation is thus quite different to many STRIPS-like planners.

### 5.4 Example: A Switch

The below represents the actions of `Switching-On` and `Switching-Off` a switch (which can have switch positions of either `*Up` or `*Down`):

```
;;; ——— Switching-On
(Switching-On has (superclasses (Action)))
;;; “The effect of a Switching-On is that the switch’s position becomes Up.”
;;; ∀s, o isa(s, Switching-On) ∧ object(s, o) →
;;;         pcs-list(s,"position(<o>,*Down)") ∧
;;;         del-list(s,"position(<o>,*Down)") ∧
;;;         add-list(s,"position(<o>,*Up)")
(every Switching-On has
  (object ((a Switch)))
  (pcs-list ((:triple (the object of Self) position *Down)))
  (del-list ((:triple (the object of Self) position *Down)))
  (add-list ((:triple (the object of Self) position *Up))))

;;; ——— Switching-Off

(Switching-Off has (superclasses (Action)))

;;; “The effect of a Switching-Off is that the switch position becomes Down.”
;;; ∀s, o isa(s, Switching-Off) ∧ object(s, o) →
;;;         pcs-list(s,"position(<o>,*Up)") ∧
;;;         del-list(s,"position(<o>,*Up)") ∧
;;;         add-list(s,"position(<o>,*Down)")
(every Switching-Off has
  (object ((a Switch)))
  (pcs-list ((:triple (the object of Self) position *Up)))
  (del-list ((:triple (the object of Self) position *Up)))
  (add-list ((:triple (the object of Self) position *Down))))
```

Let us now represent a simple the notion of a `Switch` itself, and a switch-controlled `Light`:

```
;;; “Switches are physical objects.”
;;; superclasses(Switch, Physobj)
(Switch has (superclasses (Physobj)))

;;; isa(*Up, Switch-Position) ∧ isa(*Down, Switch-Position)
(*Up   has (instance-of (Switch-Position)))
(*Down has (instance-of (Switch-Position)))

;;; “If the switch is up, then the light is bright; otherwise if it’s down the light is dark.”
;;; ∀l, s isa(l, Light) →
;;;         ( controlled-by(l, s) ∧ position(s, *Up) → brightness(l, *Bright)
;;;         ; controlled-by(l, s) ∧ position(s, *Down) → brightness(l, *Dark) )[6]
(every Light has
  (brightness ((if   ((the position of (the controlled-by of Self)) = *Up)
                then *Bright
                else (if   ((the position of (the controlled-by of Self)) = *Down)
                      then *Dark)))))
```

[6]where $(X \to Y\ ;\ Z)$ is a shorthand notation for $(X \to Y) \wedge (not(X) \to Z)$, where $not()$ is negation by failure.

```
;;; "brightness is a ramification – don't project it, recompute it for each new
;;; situation. In other words, it is a non-inertial fluent (see Section 6.2)."
(brightness has
  (instance-of (Slot))
  (fluent-status (*Fluent)))          ; i.e. a non-inertial fluent
;;; ——— Define a trivial Switch+Light circuit ———

;;; isa(*Switch1, Switch)
(*Switch1 has (instance-of (Switch)))

;;; isa(*Light1, Light) ∧ controlled-by(*Light1, *Switch1)
(*Light1 has
  (instance-of (Light))
  (controlled-by (*Switch1)))
```

This defines a light (`*Light1`) controlled by a switch (`*Switch1`). We would now like KM to reason about the effects of switching the switch up or down, including computing indirect effects (ramifications, such as the resulting brightness of the light.

One piece of information is still missing before we can start – we must first define the initial position of `*Switch1`. To do this, we create and enter a new situation (which we call the 'initial situation'). Note that we must *not* declare the initial switch position in the global situation, as this implies the switch is *always* in this position and cannot be changed. (This follows as all facts in the global situation are 'visible' to every other situation).

```
KM> (load-kb "light.km")                      ; File contains the above KB

KM> (new-situation)                           ; Create initial situation
(Changing to situation _Situation4)
(_Situation4)

;;; "In _Situation4, *Switch1 is down."
;;; holds-in(position(*Switch1, *Down), _Situation4)
[_Situation4] KM> (*Switch1 has (position (*Down)))  ; initial switch position
(*Switch1)

;;; "What is the brightness of *Light1 (in the current situation)?"
;;; { b | holds-in(brightness(*Light1, b), _Situation4) }
[_Situation4] KM> (the brightness of *Light1)   ; Example of reasoning within
(*Dark)                                         ; a situation.

;;; "Create an action of switching on *Switch1."
;;; ∃a holds-in(isa(a, Switching-On), _Situation4)∧holds-in(object(a, *Switch1), _Situation4)
[_Situation4] KM> (a Switching-On with (object (*Switch1)))
(_Switching-On7)

;;; "Compute the situation resulting from doing this action in the current situation."
;;; ι(s)( next-situation(_Situation4, s, _Switching-On7) )[7]
[_Situation4] KM> (do-and-next _Switching-On7)    ; Do it!!
(Changing to situation _Situation8)
(_Situation8)
```

[7]Here using iota notation, where $\iota(s)\alpha(s)$ denotes the unique instance for which formula $\alpha(s)$ (containing free variable $s$) is true. $\{\ \iota(s)\alpha(s)\ \} = \{\ s \mid \alpha(s)\ \}$ when $\alpha(c)$ is true of just a single instance, and is an ill-formed sentence otherwise. See [2, p47-48].

This last command to KM (`do-and-next`) causes a new situation to be created and entered. In this new situation, the `position` of `*Switch1` is now `*Up`:

```
;;; "What is the position of *Switch1 (in the new situation _Situation8)?"
;;; { p | holds-in(position(*Switch1, p), _Situation8) }
[_Situation8] KM> (the position of *Switch1)
(*Up)                                          ; Note new switch position

;;; "What is the brightness of *Light1 (in the new situation _Situation8)?"
;;; { b | holds-in(brightness(*Light1, b), _Situation8) }
[_Situation8] KM> (the brightness of *Light1)
(*Bright)                                      ; ...and its ramifications
```

Note that the direct effect of the `Switching-On` action also has the indirect effect of causing the light to become `*Bright`. KM computes these ramifications as appropriate; only the direct effects need to be recorded in the action description itself.

Note also that knowledge of the old situation (`_Situation4`) is not lost – we can return to `_Situation4` to reason about it using the `(in-situation ...)` command:

```
;;; "What was the brightness of *Light1 in _Situation4?"
;;; { b | holds-in(brightness(*Light1, b), _Situation4) }
[_Situation8] KM> (in-situation _Situation4 (the brightness of *Light1))
(*Dark)
```

### 5.5 Example: Getting and Putting

In this second example, we represent the acts of adding and removing objects from a container.

```
(Getting has (superclasses (Action)))

;;; "Getting results in the object being no longer in (ie. contents of) the source."
;;; [1] "Precondition: the object must be in the box before it can be got."
;;; ∀p isa(p, Getting) → ∃o, b isa(o, Thing) ∧ isa(b, Box) ∧ object(p, o) ∧ source(p, b) ∧
;;;         pcs-list(p,"contents(<b>,<o>)") ∧
;;;         del-list(p,"contents(<b>,<o>)")
(every Getting has
  (object ((a Thing)))
  (source ((a Box)))
  (pcs-list ((:triple (the source of Self) contents (the object of Self))))     ; [1]
  (del-list ((:triple (the source of Self) contents (the object of Self)))))

(Putting has (superclasses (Action)))

;;; "Putting results in the object being in (ie. contents of) the destination."
;;; [1] "Negated precondition: the object mustn't be in the box already!"
;;; ∀p isa(p, Putting) → ∃o, b isa(o, Thing) ∧ isa(b, Box) ∧ object(p, o) ∧ destination(p, b) ∧
;;;   ncs-list(p,"contents(<b>,<o>)") ∧ add-list(p,"contents(<b>,<o>)")
(every Putting has
  (object ((a Thing)))
  (destination ((a Box)))
  (ncs-list ((:triple (the destination of Self) contents (the object of Self))))   ; [1]
  (add-list ((:triple (the destination of Self) contents (the object of Self)))))
```

The below illustrates these actions in use:

```
KM> (*My-Box has (instance-of (Box)))                  ; Create a box...

KM> (*BlockA has (instance-of (Block)))                ; and two blocks...

KM> (*BlockB has (instance-of (Block)))

KM> (new-situation)                                    ; Enter a situation...
(_Situation194)
```

;;; "What happens (i.e., what situation results) if I put *BlockA in *My-Box?"
;;; $\iota(s)(\ \exists p\ \mathit{holds\text{-}in}(\mathit{isa}(p, \mathit{Putting}) \wedge \mathit{object}(p, *\mathit{BlockA}) \wedge \mathit{destination}(p, *\mathit{My\text{-}Box}), \_\mathit{Situation}194) \wedge$
;;; $\mathit{next\text{-}situation}(\_\mathit{Situation}194, s, p)\ )$

```
[_Situation194] KM> (do-and-next (a Putting with          ; Put *BlockA in...
                                    (object (*BlockA))
                                    (destination (*My-Box))))
(Changing to situation _Situation198)
(_Situation198)
```

;;; "What is in *My-Box now?"
;;; $\{\ c \mid \mathit{holds\text{-}in}(\mathit{contents}(*\mathit{My\text{-}Box}, c), \_\mathit{Situation}198)\ \}$

```
[_Situation198] KM> (the contents of *My-Box)
(*BlockA)                                              ; *BlockA there!
```

;;; "What happens (i.e., what situation results) if I now put *BlockB into *My-Box?"
;;; $\iota(s)(\ \exists p\ \mathit{holds\text{-}in}(\mathit{isa}(p, \mathit{Putting}) \wedge \mathit{object}(p, *\mathit{BlockB}) \wedge \mathit{destination}(p, *\mathit{My\text{-}Box}), \_\mathit{Situation}198) \wedge$
;;; $\mathit{next\text{-}situation}(\_\mathit{Situation}198, s, p)\ )$

```
[_Situation198] KM> (do-and-next (a Putting with          ; Put *BlockB in...
                                      (object (*BlockB))
                                      (destination (*My-Box))))
(Changing to situation _Situation202)
(COMMENT: Projected *My-Box contents = (*BlockA) from _Situation198 to _Situation202)
(_Situation202)
```

;;; "What is in *My-Box now?"
;;; $\{\ c \mid \mathit{holds\text{-}in}(\mathit{contents}(*\mathit{My\text{-}Box}, c), \_\mathit{Situation}202)\ \}$

```
[_Situation202] KM> (the contents of *My-Box)
(*BlockA *BlockB)                                      ; Both blocks there!
```

;;; "Suppose I now take *BlockA out?"
;;; $\iota(s)(\ \exists g\ \mathit{holds\text{-}in}(\mathit{isa}(g, \mathit{Getting}) \wedge \mathit{object}(g, *\mathit{BlockA}) \wedge \mathit{destination}(g, *\mathit{My\text{-}Box}), \_\mathit{Situation}202) \wedge$
;;; $\mathit{next\text{-}situation}(\_\mathit{Situation}198, s, g)\ )$

```
[_Situation202] KM> (do-and-next (a Getting with          ; Take *BlockA out...
                                      (object (*BlockA))
                                      (source (*My-Box))))
(Changing to situation _Situation206)
(_Situation206)
```

;;; "What is in *My-Box now?"
;;; $\{\ c \mid \mathit{holds\text{-}in}(\mathit{contents}(*\mathit{My\text{-}Box}, c), \_\mathit{Situation}206)\ \}$

```
[_Situation206] KM> (the contents of *My-Box)
(*BlockB)                                              ; Just *BlockB left
```

```
;;; "Suppose I now take *BlockB out?"
[_Situation206] KM> (do-and-next (a Getting with          ; Take *BlockB out
                                   (object (*BlockB))
                                   (source (*My-Box))))
(Changing to situation _Situation210)
(_Situation210)

;;; "What is in *My-Box now?"
;;; { c | holds-in(contents(*My-Box, c), _Situation210) }
[_Situation210] KM> (the contents of *My-Box)
NIL                                                        ; It's empty again
```

Note that in one of the steps above (labelled with the `COMMENT` message), the contents of the box were *projected* from one situation to another, reflecting the persistence of facts in the world in the absence of evidence to the contrary. We now discuss the subject of temporal projection between situations in more detail.

# 6 Temporal Projection

## 6.1 Overview

A key issue when reasoning about actions is how to deal with the "frame problem", namely how to formally state that, generally, things don't change from one situation to another, i.e. they persist. Substantial effort has been spent in AI understanding this issue. For an excellent, in-depth discussion of the frame problem, see [1].

Axioms which express this persistence are sometimes referred to as frame axioms, and the conclusion that a fact holds in situation $s'$ because it held in the previous situation $s$ (and nothing else affected it) is sometimes called the *temporal projection* of that fact from $s$ to $s'$. KM's frame axioms are not explicitly represented, but instead built into its special-purpose machinery for temporal projection. KM projects ground literals from one situation to another only if it is consistent to do so, i.e., according to the schema using Reiter's notation for normal default rules (see [3] and [4, Section 6.6]):

> "If $holds\text{-}in(prop, s)$ is true, and $holds\text{-}in(prop, s')$ is consistent with what is already known, then $holds\text{-}in(prop, s')$ is also true."

$$\forall s, s', a \; \frac{holds\text{-}in(prop, s) \wedge next\text{-}situation(s, s', a) : holds\text{-}in(prop, s')}{holds\text{-}in(prop, s')}$$

for all ground literals *prop*.

The implications of this axioms schema are computed as follows:

- For multivalued slots, all values from the previous situation will be projected from the previous situation, and then unified (using the set unification operator `&&`) with any values that slot may already have in the new situation – *except* for values which violate a constraint on that slot (For example, the value `*Fred` will not be projected if the slot in the new situation has the constraint `(<> *Fred)` on it; see the User Manual for details on stating constraints).

- For single-valued slots, any value from the previous situation will be projected and unified with any value that slot may already have in the new situation – *unless* that unification fails (i.e. it cannot be consistently unified) or it violates a constraint on that slot.

Any KM expressions on slots are evaluated *before* being projected, i.e. only the instances resulting from that evaluation are projected – the original expressions themselves are not. In other words, only ground literals are projected; rules, constraints, paths etc. are not. Also, note that computing projection is recursive: To find facts true in the previous situation, KM will look in the previous situation to the previous situation etc.
A simple example of projection is shown in the last query below:

```
KM> (new-situation)

;;; "Switch1 is down."
;;; holds-in(position(*Switch1, *Down), _Situation142)
[_Situation142] KM> (*Switch1 has (position (*Down)))

;;; "Switch1 is red."
;;; holds-in(color(*Switch1, *Red), _Situation142)
[_Situation142] KM> (*Switch1 has (color (*Red)))

;;; "What situation results if I switch Switch1 on?"
;;; ι(s)( ∃w isa(w, Switching-On)∧object(w, *Switch1)∧next-situation(_Situation142, s, w) )
[_Situation142] KM> (do-and-next
                        (a Switching-On with (object (*Switch1))))
(Changing to situation _Situation145)
(_Situation145)

;;; "What color is the switch now?"
;;; { c | holds-in(color(*Switch1, c), _Situation145) }
[_Situation145] KM> (the color of *Switch1)
(COMMENT: Projected *Switch1 color = (*Red) from _Situation142 to _Situation145)
*Red                                              ; The switch is still red!
```

When tracing KM's execution, KM by default displays that it is computing a slot's value in a previous situation, but not the details of that computation. To have KM show those details, switch on "tracing in other situations" using the `+S` option at one of the trace points.

## 6.2 Controlling Projection: Inertial and Non-Inertial Fluents

As we discussed earlier in Section 3.3, projection is not appropriate for all slots. In particular, *ramifications* (indirect effects of actions) should not be projected; rather, they should be recomputed from the (possibly projected) direct effects of actions in each new situation, in case some of those direct effects no longer hold.

As Section 3.3 described, we can tell KM whether a slot's values should be projectable or not by setting its `fluent-status` to either `*Inertial-Fluent` (projectable), `*Fluent` (non-projectable, but still varying between situations), and `*Non-Fluent` (values are the same in all situations). Direct effects of actions should be `*Inertial-Fluents`, ramifications (indirect effects) should be `*Fluents`, and slots whose values are constant should be `*Non-Fluents`. Values of non-fluent slots are stored in the `*Global` situation, and thus always visible in all situations (so do not need to be "projected").

| slot's fluent-status | Meaning |
|---|---|
| *Inertial-Fluent | Use this for direct effects of actions (and direct effects introduced by the user). Values will be projected from one situation to the next, if consistent to do so. |
| *Fluent | Use this for indirect effects (ramifications) of actions. Values of this slot should be recomputed in each new situation. |
| *Non-Fluent | Use this for slots whose values will not vary between situations. |

A slot's fluent status is declared by setting its `fluent-status` attribute in the slot declaration:

```
;;; "Values for possible-actions should not be projected."
KM> (possible-actions has
      (instance-of (Slot))
      (fluent-status (*Fluent)))
```

By default, all slots are `*Inertial-Fluents`. This default can be changed using the `default-fluent-status` command, described earlier in Section 3.3.

Note that declaring slots for ramifications as `*Inertial-Slots` is not quite an error – rather, it will cause KM to do extra work, and could lead to problems if the user has not declared all the constraints on the KB. For example, if `best-move` in a chess game is a ramification (computable from the direct piece positions), but we accidentally declare it as an `*Inertial-Fluent` rather than a `*Fluent`, then KM will think the `best-move` for any position is the current best move *and* all the projected, previous best moves from earlier situations in the game (i.e. there are multiple best moves). We could fix this by constraining `best-move` to have only one value (by declaring `(best-move has (cardinality (N-to-1)))`), but then KM will wonder if all these different moves may be coreferential, and try and unify them together! We could add in further constraints to avoid this unification succeeding (e.g. requiring the `destination` of a move to be a unique square on the board), and then we will have the result we want: the unification will fail, and thus the projection will not succeed. But this is a rather long-winded way of blocking the projection that we could have avoided in the first place by simply declaring `best-move` as a `*Fluent` (i.e., non-inertial fluent).

# 7 Testing the Preconditions of Actions

As described earlier in Section 5.3, the `pcs-list` slot of an action declares facts which are necessarily true in the situation immediately preceding the performance of that action. Similarly, the `ncs-list` declares those which are necessarily false. If our knowledge-base is designed to have complete knowledge of these facts (i.e., so that failure to prove them implies they are false), then we can also use the precondition list to *test* whether an action is indeed possible in a particular situation. To do this, we can apply KM's operators (`all-true` *expr*), and (`some-true` *expr*), described earlier in Section 5.1.2, to the preconditions and negated preconditions of an actions, to see if they hold. This can be conveniently done by creating something like an `is-possible` slot on actions, which tests the preconditions and returns a non-nil value if they are all satisfied:

```
;;; "An action is possible if its preconditions are true, and its negated preconditions are false."
KM> (every Action has
      (is-possible ((    (all-true (the pcs-list of Self))
                     and (not (some-true (the ncs-list of Self)))))))
```

Before proceeding, however, there are two complications which we must take care of. First, as `is-possible` is a *ramification* of the state of the world, its value should not be projected from situation to situation, and instead recomputed afresh in each situation. To do this, we declare it as a non-inertial fluent, by setting its `fluent-status` slot to `*Fluent` (also see Section 6.2). Second, in general, the preconditions of an action may be situation-dependent (see Section 5.1.5), and thus there may be no universally true preconditions, i.e., asking for the preconditions in the global situation will return the empty set `()`. However, we do not want KM to therefore conclude, in the global situation, that these (zero) preconditions are universally satisfied. To do this, KM should avoid computing the value of `is-possible` in the global situation, to circumvent this problem with negation as failure. We can do this by making this slot `situation-specific` (See Section 3.4):

```
;;; "is-possible is a non-inertial fluent, and has no meaningful global value."
KM> (is-possible has
       (instance-of (Slot))
       (fluent-status (*Fluent))
       (situation-specific (t)))
```

We can now combine this with the definitions of `Getting` and `Putting` from Section 5.5 earlier, to compute which action(s) are actually doable in a particular situation:

```
;;; Define some initial objects
KM> (*My-Box has (instance-of (Container)))

KM> (*BlockA has (instance-of (Block)))

KM> (*BlockB has (instance-of (Block)))

KM> (new-situation)

[_Situation29] KM> (the contents of *My-Box)
NIL                                          ; *My-Box is initally empty

;;; "For all the potential actions I can do, which are actually possible?"
[_Situation29] KM>
  (allof (:set (a Getting with (object (*BlockA)) (source (*My-Box)))
               (a Getting with (object (*BlockB)) (source (*My-Box)))
               (a Putting with (object (*BlockA)) (destination (*My-Box)))
               (a Putting with (object (*BlockB)) (destination (*My-Box))))
   where (the is-possible of It))
(_Putting32 _Putting33)                      ; Can put *BlockA or *BlockB in *My-Box

;;; Do the first put action, putting *BlockA in the box...
[_Situation29] KM> (do-and-next _Putting32)
(Changing to situation _Situation40)
(_Situation40)

;;; Check it's there...
[_Situation40] KM> (the contents of *My-Box)
(*BlockA)

;;; "Now which actions are possible?"
[_Situation40] KM> (allof (:set
```

```
                              (a Getting with (object (*BlockA)) (source (*My-Box)))
                              (a Getting with (object (*BlockB)) (source (*My-Box)))
                              (a Putting with (object (*BlockA)) (destination (*My-Box)))
                              (a Putting with (object (*BlockB)) (destination (*My-Box))))
                        where (the is-possible of It))
(_Getting41 _Putting44)                         ; Either you can get *BlockA out, or
                                                ; put *BlockB back in.
```

In this example, we have rather crudely enumerated all the potential actions which we want KM to consider. We show later in Section 9 how this set can be computed automatically instead.

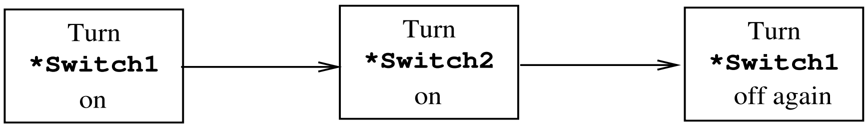
# 8 Simulation

## 8.1 Introduction

The `do-and-next` command simulates the execution of a single action, and causes KM to move to a new situation (representing the post-action state of the world). We can thus straightforwardly simulate the execution of a **plan**, eg. (here) a sequence of actions, by repeatedly performing a `do-and-next` operation. Note that (unlike a STRIPS planner) the intermediate situations during execution are retained in the KB and are still accessible – we can thus ask questions about the entire execution eg. “Did light 2 ever go on?”.

## 8.2 Example: An Electrical Circuit

Consider the simple representation of an electrical circuit fragment, shown in Figures 3 and 4. This describes two lights connected to two switches, and the actions `Switching-On` and `Switching-Off`. Now, we can represent a `Plan` as a sequence of actions, for example the plan:




could be represented as below. (Here we assume KM will preserve action ordering. More generally, something like a `next-action` slot should be used to relate actions in a sequence):

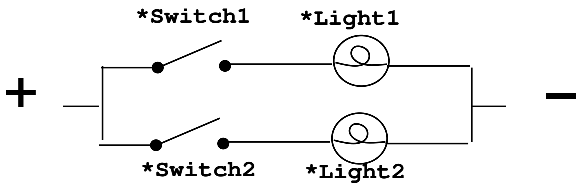
```
(*My-Plan has
  (instance-of (Plan))
  (subevents (
    (a Switching-On with (object (*Switch1)))         ; step 1
    (a Switching-On with (object (*Switch2)))         ; step 2
    (a Switching-Off with (object (*Switch1)))        ; step 3
      )))
```

We can execute the plan by manually stepping through each of the three action steps:

```
;;; Find the subevents in the plan...
KM> (the subevents of *My-Plan)
(_Switching-On2 _Switching-On3 _Switching-Off4)          ; The three steps

KM> (new-situation)                                      ; Define initial situation
[_Situation5] KM> (*Switch1 has (position (*Down)))
```

```
#| =============== START OF DEMO KB ===============
```




```
|#
;;; “*My-Circuit has two lights and two switches.”
(*My-Circuit has
  (instance-of (Circuit))
  (switches (*Switch1 *Switch2))
  (lights (*Light1 *Light2))
  (parts ((the lights of Self) (the switches of Self))))

;;; “*Light1 is controlled by *Switch1.”
(*Light1 has
  (instance-of (Light))
  (controlled-by (*Switch1)))

;;; “*Light2 is controlled by *Switch2.”
(*Light2 has
  (instance-of (Light))
  (controlled-by (*Switch2)))

(*Switch1 has (instance-of (Switch)))
(*Switch2 has (instance-of (Switch)))

;;; ——— Ramifications (non-inertial fluents) ———
(illuminated-lights has (instance-of (Slot)) (fluent-status (*Fluent)))
(brightness         has (instance-of (Slot)) (fluent-status (*Fluent)))
(is-possible        has (instance-of (Slot)) (fluent-status (*Fluent)))
(possible-actions   has (instance-of (Slot)) (fluent-status (*Fluent)))
(applicable-actions has (instance-of (Slot)) (fluent-status (*Fluent)))

;;; ——— Circuit ———
(Circuit has (superclasses (Physobj)))

;;; “The illuminated lights in a circuit are all those which are bright.”
(every Circuit has
  (illuminated-lights (
        (allof (the lights of Self)
         where ((the brightness of It) = *Bright)))))

;;; “If a light’s switch is *Up, then the light is on (ie. is *Bright).”
(every Light has
  (brightness ((if   ((the position of (the controlled-by of Self)) = *Up)
                then *Bright
                else (if   ((the position of (the controlled-by of Self)) = *Down)
                      then *Dark)))))
```

Figure 3: A simple representation of an electrical circuit (Part 1).

```
;;; ——— Define a test of preconditions for actions ———

(every Action has
  (is-possible ((    (all-true (the pcs-list of Self))
                 and (not (some-true (the ncs-list of Self)))))))

;;; ——— Switching-On, Switching-Off ———

;;; Define Switching-On and Switching-Off actions...
(Switching-On has (superclasses (Action)))

;;; "Switching-On causes the switch to move *Up. Can only do this if the
;;; switch position is initially *Down."
(every Switching-On has
  (object ((a Switch)))
  (pcs-list ((:triple (the object of Self) position *Down)))
  (del-list ((:triple (the object of Self) position *Down)))
  (add-list ((:triple (the object of Self) position *Up))))

(Switching-Off has (superclasses (Action)))

(every Switching-Off has
  (object ((a Switch)))
  (pcs-list ((:triple (the object of Self) position *Up)))
  (del-list ((:triple (the object of Self) position *Up)))
  (add-list ((:triple (the object of Self) position *Down))))

;;; ——— Switch ———

(Switch has (superclasses (Physobj)))

;;; Things you can potentially do to a switch: Switching-On and Switching-Off.
(every Switch has
  (applicable-actions (
    (a Switching-Off with (object (Self)))
    (a Switching-On  with (object (Self))))))

;;; ——— Physobj ———

;;; "The actions I can perform on a physical object are
;;; [1] those applicable actions which are also possible
;;; [2] (recursively) the actions I can perform on its parts."
(every Physobj has
  (possible-actions (
     (allof (the applicable-actions of Self)
      where (the is-possible of It))                         ; [1]
     (the possible-actions of (the parts of Self)))))         ; [2]

;;; ====== END ======
```

Figure 4: A simple representation of an electrical circuit (Part 2).

```
[_Situation5] KM> (*Switch2 has (position (*Down)))

[_Situation5] KM> (do-and-next _Switching-On2)           ; Do step 1
(Changing to situation _Situation6)

[_Situation6] KM> (do-and-next _Switching-On3)           ; Do step 2
(Changing to situation _Situation7)

[_Situation7] KM> (do-and-next _Switching-Off4)          ; Do step 3
(Changing to situation _Situation8)

;;; "Which lights are now on?"
;;; { l | holds-in(illuminated-lights(*My-Circuit, l), _Situation8) }
[_Situation8] KM> (the illuminated-lights of *My-Circuit) ; Result?
(*Light2)                                                 ; Just *Light2 on
```

More concisely, we can execute the plan with a single command using a `forall` expression:

```
KM> (new-situation)                                         ; Define initial state
[_Situation237] KM> (*Switch1 has (position (*Down)))

[_Situation237] KM> (*Switch2 has (position (*Down)))

;;; Now do all the subevents!
[_Situation237] KM> (forall (in-situation *Global (the subevents of *My-Plan))
                              (do-and-next It))
(Changing to situation _Situation239)
(Changing to situation _Situation241)
(Changing to situation _Situation243)

;;; "Which lights are now on?"
;;; { l | holds-in(illuminated-lights(*My-Circuit, l), _Situation8) }
[_Situation243] KM> (the illuminated-lights of *My-Circuit) ; Result?
(COMMENT: Projected *Switch2 position = (*Up) from _Situation241 to _Situation243)
(*Light2)                                                  ; Just *Light2 on
```

In this example, the `forall` expression iterates over the steps of the plan, executing each one in order using the `do-and-next` command.

### 8.3 Quantifying Over Intermediate Situations

In addition to simulating the plan execution, we can quantify over the intermediate situations that were created:

```
;;; "Are the switches ever both up at the same time?"
;;; (i.e., "Does there exist a Situation in which both *Switch1 is *Up and *Switch2 is *Up?")
;;; oneof({ s | isa(s, Situation) ∧
;;;            holds-in(position(*Switch1, *Up), s) ∧ holds-in(position(*Switch2, *Up), s) })
KM> (oneof (the all-instances of Situation)
     where (in-situation It (    ((the position of *Switch1) = *Up)
                              and ((the position of *Switch2) = *Up)))))
(COMMENT: Projected *Switch2 position = (*Down) from _Situation237 to _Situation239)
(COMMENT: Projected *Switch1 position = (*Up) from _Situation239 to _Situation241)
(_Situation241)                    ;;; Answer: Yes! (both are *Up in _Situation241)
```

Instead of explicitly listing the intermediate situations, we could ask KM to compute them, eg.:

;;; Define future-situations as the transitive closure of next-situation.
;;; $\forall s, s'\ isa(s, Situation) \wedge next\text{-}situation(s, s') \rightarrow future\text{-}situations(s, s')$
;;; $\forall s, s', s''\ isa(s, Situation) \wedge next\text{-}situation(s, s') \wedge future\text{-}situations(s, s'') \rightarrow future\text{-}situations(s, s'')$

```
KM> (every Situation has
        (future-situations ((the next-situation of Self)
                            (the future-situations of (the next-situation of Self)))))
```

;;; "What are the future situations of Situation237?"
;;; $\{\ s \mid future\text{-}situations(\_Situation237, s)\ \}$

```
KM> (the future-situations of _Situation237)
(_Situation239 _Situation241 _Situation243)
```

;;; "Are the switches ever both up at the same time?"
;;; (Here we just name the initial situation, _Situation237)
;;; $oneof(\{\ s \mid s \in \{\_Situation237\} \cup \{s' | future\text{-}situations(\_Situation237, s')\}$
;;; $\wedge\ holds\text{-}in(position(*Switch1, *Up), s) \wedge holds\text{-}in(position(*Switch2, *Up), s)\ \})$

```
KM> (oneof (:set _Situation237 (the future-situations of _Situation237))
     where (in-situation It (    ((the position of *Switch1) = *Up)
                             and ((the position of *Switch2) = *Up))))
(_Situation241)                        ;;; Answer: Yes!
```

# 9 Possible Worlds and Envisionments

## 9.1 Introduction

As well as simulating execution of a plan, which traces out a single chain of events in the world, we can have KM explore possible, alternative events which could occur, and the alternative consequent situations. As KM retains knowledge about these different situations, we can then ask questions such as "Is it possible that X?" (ie. "For all possible future situations, does there exist one in which X?"), or "What outcome does X hope for by doing A?" (ie. "For all possible outcomes of A, which one is most preferred by X?"). Answering these questions requires KM to construct and reason about an **envisionment** of the world – namely a graph of possible future situations. This is similar to the creation and use of envisionments in qualitative reasoning, except in this case each situation is represented as a first-order logic (ie. KM) theory, rather than a set of parameter values.

## 9.2 Creating Possible, Alternative Situations

To explore alternative situations which could arise from performing alternative actions in some situation $S$, we can ask KM to try `do`ing different actions from the same starting situation. These alternative next situations are stored on the `next-situation` slot of the current situation frame, with a different action associated with each (bundled as a `(:args ...)` structure, as described earlier in Section 5.2). For example, continuing with the example circuit KB in Section 8, we might ask KM to consider both actions possible from the initial state (namely switching `*Light1` on, and switching `*Light2` on):

```
KM> (new-situation)                      ; Define initial situation

[_Situation7] KM> (*Switch1 has (position (*Down)))
```

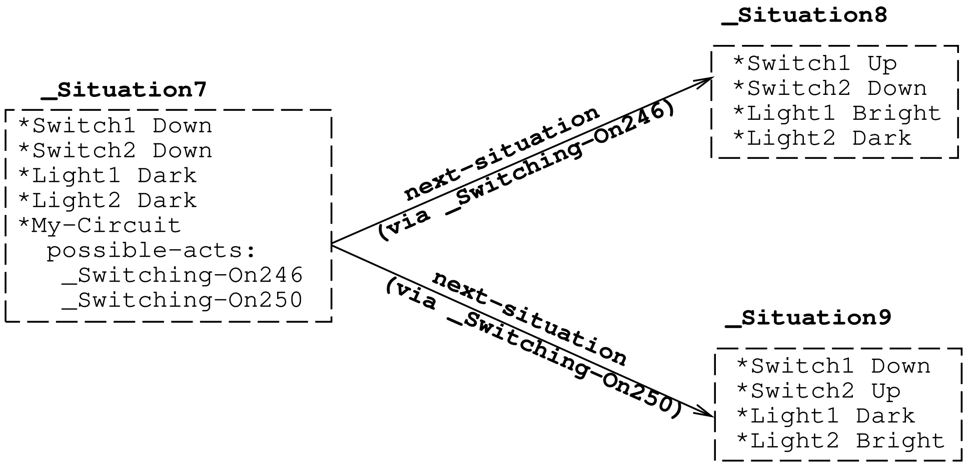



Figure 5: Creating and exploring possible futures (see Section9.2).

```
[_Situation7] KM> (*Switch2 has (position (*Down)))
```

;;; "What possible things can I do from here?"
;;; { $a$ | $holds\text{-}in(possible\text{-}actions(*My\text{-}Circuit, a), \_Situation7)$ }

```
[_Situation7] KM> (the possible-actions of *My-Circuit)
(_Switching-On246 _Switching-On250)
```

;;; "Which situation(s) arise from doing these action(s)?"
;;; { $s$ | $\forall a\ holds\text{-}in(possible\text{-}actions(*My\text{-}Circuit, a), \_Situation7) \wedge next\text{-}situation(\_Situation7, s, a)$ }

```
[_Situation7] KM> (forall (the possible-actions of *My-Circuit)
                          (do It))
(Changing to situation _Situation8)
(Changing to situation _Situation7)
(Changing to situation _Situation9)
(Changing to situation _Situation7)
(_Situation8 _Situation9)
```

;;; Now note the multiple next situations, and actions which led to them,
;;; which were created. (See Section 5.2 for details).

```
[_Situation7] KM> (showme _Situation7)          ; Note: Still in initial situation
(_Situation7 has
  (next-situation ((:args _Situation8 _Switching-On246)
                   (:args _Situation9 _Switching-On250))))
```

Note above that using the `do` command, rather than `do-and-next` command, means that KM *doesn't* change to the new situation after performing the action. Thus each new situation is created as an alternative future of the current situation. Had we used `do-and-next` instead, KM would have created a single sequence of situations, corresponding to applying each action one after the other. The situations created and their relationships are shown in Figure 5. We can now query this simple one-step envisionment:

;;; "What are the possible futures of this situation?"
;;; (Note, `the1` extracts just the situation from the (`:arg` *sitn action*) bundle)
;;; { $s$ | $\forall a\ next\text{-}situation(\_Situation7, s, a)$ }

```
[_Situation7] KM> (the1 next-situation of _Situation7)
```

```
(_Situation9 _Situation8)
```

;;; "Is *Light1 on in Situation8?"
;;; $\{ \; b \mid \mathit{holds\text{-}in}(\mathit{brightness}(*\mathit{Light}1, b), \_\mathit{Situation}8) \; \}$

```
[_Situation7] KM> (in-situation _Situation8 (the brightness of *Light1))
(*Bright)                                                ; (i.e., yes)
```

;;; "Is *Light1 on in Situation9?"
;;; $\{ \; b \mid \mathit{holds\text{-}in}(\mathit{brightness}(*\mathit{Light}1, b), \_\mathit{Situation}9) \; \}$

```
[_Situation7] KM> (in-situation _Situation9 (the brightness of *Light1))
(*Dark)                                                  ; (i.e., no)
```

;;; "Is there a possible future in which *Light1 is *Bright?"
;;; $\mathit{oneof}(\{ \; s \mid \forall a \; \mathit{next\text{-}situation}(\_\mathit{Situation}7, s, a) \land \mathit{holds\text{-}in}(\mathit{brightness}(*\mathit{Light}1, *\mathit{Bright}), s) \; \})$

```
KM> (oneof (the1 next-situation of _Situation7)
     where (in-situation It ((the brightness of *Light1) = *Bright))))
(_Situation8)                                     ; Yes!
```

;;; "What do I do to achieve it?" (i.e., "Which action created this situation?")
;;;, $\iota(a)( \; \forall s \; \mathit{prev\text{-}situation}(\_\mathit{Situation}8, s, a) \; )$

```
KM> (the2 prev-situation of _Situation8)
(_Switching-On1)                                  ; Answer: Do _Switching-On1
```

;;; "Which switch does Switching-On1 switch on?"
;;; $\{ \; o \mid \mathit{holds\text{-}in}(\mathit{object}(\_\mathit{Switching\text{-}On}1, o), \_\mathit{Situation}7) \; \}$

```
KM> (in-situation _Situation7 (the object of _Switching-On1))
(*Switch1)
```

In this example, a single command (annotated above as "What are the possible futures?") issued from the global situation caused KM to create the (one-step-deep) envisionment. We can then quantify over those possible futures to see if `*Light1` is ever `*Bright`. Again note this is beyond the capabilities of a STRIPS-like reasoner in two ways: First we are quantifying over possible futures, and second we are querying about *indirect* effects of actions (`Switching-On`'s only direct effect is to move the switch – inference is required to conclude that therefore the light will become on).

# 10 Existence, and Actions which Create and Destroy

## 10.1 Introduction

Some actions (e.g., baking a cake) have the effect of creating or destroying entities. Modeling these kinds of actions raises some perplexing issues concerning the nature of existence. What does it mean to say something exists (or no longer exists)? In language, it seems we can talk about non-existent objects (""My pet dragon"), objects which no longer exist ("Alan Turing was a brilliant mathematician"), and objects which never quite existed ("The strike was averted"). And, seemingly paradoxically, it seems we have to always assert the existence ($\exists$) of such non-existent objects before we can say anything about them in logic.

There are several alternative philosophies about this (see Hirst's paper [5] for an insightful exposition and discussion of these), but for simplicity we describe just two, drawing heavily from the Hirst paper. What sort of objects should we allow in our universe of discourse, i.e., what sort of things should we allow the logical quantifier '$\exists$' to quantify over? Or in KM terms, what instances should actually allow the creation of? The first philosophy, called the Russell-Quine position, only allows 'real' objects into this universe, such as houses and trees, but not dragons because they don't exist (although Russell and Quine would, we assume, allow the *idea* of dragons to exist). Thus "dragons don't exist" would be expressed as (say) $\neg\exists x\ dragon(x)$. Their claim was that natural language statements apparently about non-existent entities can always be reformulated in a way which does not require their denotation (as, after all, they don't exist). We can count how many houses (dragons, etc.) there are in the world (which our theory describes) by counting how many (non-coreferential) constants are in that theory with the property of interest.

The second philosophy is one which Hobbs refers to as "ontological promiscuity" [6], proposed by various authors in different forms [5]. This approach allows 'everything' to be in the universe of discourse, including unreal objects such as dragons, and then treats existence (in the colloquial form) to be just another property of an object, denoted for example using a predicate (e.g. $exists(x)$). This approach makes a clear distinction between logical existence ($\exists$) and common-sense existence, the former simply meaning that we can talk about it and nothing more, while the latter is a short-hand for a whole collection of properties (e.g. has mass, physical location). So what are these properties? In fact, as Hirst shows [5], (common-sense) existence can mean many different things depending on what we are talking about, and so he proposes an ontology of different 'existence predicates' to capture these different nuances.

Our approach in KM follows this second philosophy[8]. In fact, when we start talking about situations and changes with time (where objects can come and go from material existence), it is difficult to see how to avoid denoting such objects in situations where they no longer exist (one cannot "destroy a constant"). However, if we do admit non-existent objects into the universe, we then need to annotate objects as to their existence status. One approach to this (as Hobbs took) is to ascribe an existence property to every object, e.g. using a slot like `existence-status` with values such as `*Real`, `*Imaginary`, etc. This seems workable, although also rather cumbersome as we now need to check and propagate the existence status of objects through every action (e.g. baking a cake: "If I have some flour, and the flour exists, and I have some eggs, and the eggs exist, and.... then this will result in a cake, and the cake will exist, and the flour will no longer exist, and the eggs will no longer exist, and ...."). It might be possible to reduce this problem by expressing these constraints at a more general level (e.g. "Every physical action requires that all its agents, patients, and instruments exist as real objects, as preconditions for that action to occur."). Also, when we ask "How many X's are there?", we need to be clear whether X includes immaterial, hypothetical etc. objects, as well as material objects.

In fact, using an existence predicate (slot) can still lead to confusion. What exactly does it mean to have an `existence-status` of `*Real`? Is Alan Turing no longer 'real' after he died? We can still talk about him, and relate him to other objects (e.g., "Sue loves Alan Turing"). "Going out of existence" seems to mean that some, but not all, propositions about an object become false – and the choice of which those are seems to vary. If my house is destroyed it ceases to have material properties, but still retains other properties (e.g., date of construction). An alternative approach (which may be no better) is to avoid mentioning "existence" at all, and instead, as for any other action, list the specific properties which become true/false after a creation or destruction action is performed. This approach views 'existing' simply as a shorthand for a whole collection of properties (e.g., physical location). Creating and destroying objects now explicitly adds and deletes these properties for objects, just like any other property. A precondition

[8]In fact, as KM answers questions through the creation and reasoning about instances, we have already deviated from the Russell-Quine position as these instances do not necessarily denote real-world objects. Rather, they are hypothetical objects which KM is constructing in order to answer queries.

that an object 'exists' now becomes a precondition that it has the physical properties that we care about (e.g., location). This approach avoids confusion about the many meanings of 'existence' by forcing us to be explicit, which itself requires making some modeling decisions (e.g. can an imaginary object have mass? And if not, we must be careful not to make statements such as `(every Airplane has (mass (...)))`, where the class `Airplane` covers both real and imaginary airplanes). Davis takes something like this approach in his theory of cutting [2], where he uses the predicate $material(x)$ (meaning "is physically material"), rather than $exists(x)$, to be explicit about what property the cutting is affecting.

### 10.2 Example: Baking a Cake

Below is a simple representation of baking a cake, which uses the slot `material` to denote whether the object is physically material or not. It asserts that the ingredients are material before the action (but the cake is not), and that the cake is material afterwards (but the ingredients are not). Note that we can still refer to the ingredients and the cake in any situation, they just may not be physically material in those situations.

```
;;; [1] preconditions: The ingredients must exist before the baking
;;; [2] negated preconditions: The cake mustn't exist before the baking
;;; [3,4] effects: The material status of the objects change.
KM> (every Baking has
      (agent ((a Person)))
      (ingredients ((must-be-a Food)))
      (result ((must-be-a Food)))          ; (sometimes violated in practice... :-))
      (pcs-list ((forall (the ingredients of Self)
                         (:triple It is-material t))))                   ; [1]
      (ncs-list ((:triple (the result of Self) is-material t)))          ; [2]
      (add-list ((:triple (the result of Self) is-material t)))          ; [3]
      (del-list ((forall (the ingredients of Self)
                      (:triple It is-material t)))))                     ; [4]

KM> (Baking-A-Cake has (superclasses (Baking)))

KM> (every Baking-A-Cake has
      (ingredients ((a Piece-Of-Flour) (a Piece-Of-Sugar) (a Piece-Of-Butter)
                    (a Egg) (a Egg)))
      (result ((a Cake))))

KM> (*Pete has (instance-of (Person)))

KM> (new-situation)

;;; Create a baking action...
[_Situation0] (a Baking-A-Cake with
                  (agent (*Pete)))

;;; Do the action...
[_Situation0] (do-and-next (the Baking-A-Cake))

;;; "Is the cake physically material now?"
[_Situation1] KM> (the is-material of (the result of (thelast Baking-A-Cake)))
(t)
```

```
;;; "Is the flour physically material now?"
[_Situation1] KM> (the is-material of
                        (the Piece-Of-Flour ingredients of (thelast Baking-A-Cake)))
NIL

;;; "Was the flour physically material before?"
[_Situation1] KM> (in-situation (the prev-situation of (curr-situation))
                                (the is-material of
                                  (the Piece-Of-Flour ingredients of
                                    (thelast Baking-A-Cake))))
(t)
```

Note that in the above example, the cake is visible in all situations, but has no physical properties (e.g. being owned by someone, having a mass) until it has materialized in the oven.

### 10.3 Example: The Magician's Rabbit

We now suggest an alternative approach, also worthy of consideration. In this approach, we consider the class `Thing` to include everything (real, imaginary, to-be-real, was-real), while normal classes like `Rabbit` only refer to physically material objects. Thus destroying a rabbit (say) is modeled by removing it from the class `Rabbit`, although it (the symbol denoting the ex-rabbit) is now just in the class `Thing`. Similarly, creating a rabbit creates an instance in class `Rabbit` in the situation in which it was created, while prior to that the to-be-a-rabbit is merely in class `Thing`. This is illustrated below, in which we model a (real) magician creating a rabbit out of thin air, turning it into a dove, and then making it disappear again. Throughout the scenario, the symbol `*MyThing` denotes the identity of this object, including before and after it materially exists.

```
KM>  (new-situation)

;;; "Create a rabbit." (Set *MyThing's class to Rabbit).
[_Situation0] KM>  (do-and-next
                     (a Create with
                       (created (*MyThing))
                       (will-be-a (Rabbit))
                       (add-list ((:triple (the created of Self)
                                           instance-of
                                           (the will-be-a of Self))))))
(COMMENT: Changing to situation _Situation2)

;;; "Now change it into a dove." (Changing *MyThing's class from Rabbit to Dove).
[_Situation2] KM>  (do-and-next
                     (a Change with
                       (changed (*MyThing))
                       (will-be-a (Dove))
                       (del-list ((:triple (the changed of Self) instance-of
                                           (the instance-of of (the changed of Self)))))
                       (add-list ((:triple (the changed of Self) instance-of
                                           (the will-be-a of Self))))))
(COMMENT: Changing to situation _Situation4)

;;; "Now make it disappear again." (Changing *MyThing's class from Dove to Thing).
```

```
[_Situation4] KM>  (do-and-next
  (a Destroy with
     (destroyed (*MyThing))
     (del-list ((:triple (the destroyed of Self) instance-of
                         (the instance-of of (the destroyed of Self)))))))
(COMMENT: Changing to situation _Situation6)

;;; "Print out what *MyThing was in each situation in the sequence."
[_Situation6] KM> (forall (the instances of Situation)
                    (km-format t "In ~a, *MyThing was a ~a.~%"
                               It (in-situation It (the instance-of of *MyThing))))
In (_Situation0), *MyThing was a (Thing).
In (_Situation2), *MyThing was a (Rabbit).
In (_Situation4), *MyThing was a (Dove).
In (_Situation6), *MyThing was a (Thing).
```

The advantage of this approach is that it avoids repeatedly testing an existence status predicate, as objects in subclasses of Thing are considered to materially exist already (i.e., class membership tests implicitly test existence also). However, it does require the inference engine to tolerate dynamically changing classes (which KM does). Further experimentation is needed to decide if this approach is most suitable.

# 11 Some Limitations

## 11.1 Chronological Minimization

KM's projection algorithm implements what is referred to as "temporal minimization"; that is, it assumes that things won't change in the world until explicitly told otherwise (thus "minimizing" the changes with respect to time). This assumption, however, is not always valid. Consider the following "stolen car" scenario [1, p79]: I park my car, then I go and do some things, then I come back and my car is gone (someone stole it). When was my car stolen? Using chronological minimization, KM will project the location of my car (i.e., in the car park) through all the situations until the one when I return and find my car missing, and thus it will conclude that my car was stolen at just the moment before I returned, clearly too strong a conclusion. Lifschitz and others have studied this problem and suggested an alternative set of assumptions for reasoning about actions, based instead on "causal minimization" (minimize the unexplained changes in a scenario, rather than the temporal changes), which offers a solution to this problem. Further discussion about these approaches can be found in [1, Chapter 5].

## 11.2 Projection Back in Time

It is often valid to project information *backwards* in time, as well as forwards. For example, if my cup is on the table, then I should be able to conclude that 10 minutes ago it was also on the table (if I know that no intervening actions occurred which affected its location). KM's projection mechanism, however, does not currently project information back in time in this way, and thus will miss such conclusions. If it were to do so, it would need some policy for handling cases where forward-projected and backward-projected information conflicted (for example, as would happen in the stolen car scenario in Section 11.1). Some formalisms for the event calculus (e.g. [7]) have explored adding this capability.

### 11.3 Disjunctive Ramifications and Multiple Possible Extensions

We stated earlier (Section 5.1.6), that an action representation need only describe its 'direct' effects. Then, rules could be used to compute the 'indirect' effects (ramifications). While this approach can greatly simplify describing actions, it can also introduce ambiguity as to what the result of an action might be. For example, suppose $a$ and $b$ are true in situation $S_1$, then an action is performed which makes $a$ and $b$ mutually exclusive in situation $S_2$: now there are two equally valid, possible models for $S_2$, namely $\{a, \neg b\}$ and $\{\neg a, b\}$. Which should be preferred? As a concrete illustration, suppose Rachel (age 4) has dual nationalities of British and Swiss, but then the "event" of her reaching the age of 18 occurs (when, let's say, she must choose one of these). The direct effect of the action is that her age becomes 18. But the ramification is now that if her nationality is still British then she will also not be Swiss, and similarly if she is still Swiss then she will also not be British. Two possible extensions of the new situation are possible. This phenomenon only arises when actions have disjunctive ($a \vee b$) effects – and as KM does not allow the direct effects of actions to be disjunctive, it only occurs in KM when ramifications are involved (allowing an effect to have some disjunctive ramifications).

KM does not have special purpose machinery for handling this phenomenon, and will instead make an arbitrary choice as to which extension it constructs, based on which literals are projected first (in turn determined by which questions are asked the system). In the above situation where values become mutually inconsistent, KM may also project neither value, or (inconsistently) both values, depending on how the rest of the KB is constructed and which queries are issued. Disjunctive ramifications are probably best avoided!

### 11.4 Modeling Continuous Change

One inherent limitation of situation calculus is that it is not easy to model continuously changing processes (e.g., filling a bath tub), although there has been some work looking at ways in which this can be done (e.g., [8]). Of course, continuous change can always be shoe-horned into discrete actions (e.g., the bath changes from 'empty' to 'full'), but this loses some of the information that we may care about, in particular those related to time (e.g., "How full is the bath at time t?"). Continuous processes are probably better modeled using qualitative modeling (e.g., [9]), or the event calculus (e.g., [1]), rather than situation calculus. For example the qualitative model might be specified using KM (e.g. using slots such as `Q+`, `Q-`), and then an external qualitative simulator would then reason with that model for answering questions. This approach was taken in the TRIPEL system for the Botany Knowledge-Base project [10].

### 11.5 The Situation-Action Dichotomy

Traditional Situation Calculus forces the user to divide the world into "facts" and "actions". This division sometimes feels awkward, as in natural language we frequently treat an action as an instantaneous event in one sentence, and then as an ongoing situation in another. For example, "John flew from Boston to Seattle" suggests an action which deletes John being in Boston and adds John being in Seattle, while "During the flight he slept" suggests a situation in which the flying is on-going. Consider, which of the following are actions: flying, eating, growing, eating, being? It seems clear that the answers are situation- and task-dependent.

There is a modified approach we can adopt to avoids this dichotomy, namely to also model the situation *during* an action occurring. Situation Calculus normally ignores this "during-situation", only modeling the "before-situation" and "after-situation". If we adopt this modified approach, however, then we must also flag the (reified instances denoting) actions as "happening" in those "during situations', and then not happening in the situation when the action is finished. In this way, the fact that an action is ongoing can be treated just like any other fact, and similarly every other fact can be thought of as being a kind of "ongoing action", e.g., "Sue loves John" is a (probably rather long) during-situation of some wider event.

For example, "John flew from Boston to Seattle" now becomes an action which deletes that John is in Boston (i.e. John is in Boston in the before-situation, but not after), and adds that John is in Seattle (i.e. John is in Seattle in the after-situation, but not before), but *also* creates a during-situation in which this flying action (`_Flying231`, say), has the value `t` for its `is-ongoing` (say) slot (but not in the preceding or subsequent situations). KM can now reason about the during-situation, including implications of flying being ongoing. We could also break down the during-situation further into subsituations (Section 3), if we were interested in modeling different parts of the flying activity (e.g., takeoff, cruising, landing).

This extended representation of actions would require a small extension to KM, in which a new `do` command (e.g. `do-with-during`) was introduced which explicitly created this during-situation. This has not been implemented yet but could be quite easily should there be sufficient demand and interest.

## References


[1] Murray Shanahan. *Solving the Frame Problem*. MIT Press, Apr 1997.

[2] Earnest Davis. *Representations of Commonsense Knowledge*. Kaufmann, CA, 1990.

[3] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.

[4] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations for Artificial Intelligence*. Kaufmann, CA, 1987.

[5] Graeme Hirst. Existence assumptions in knowledge representation. *Artificial Intelligence*, 49:199–242, 1991.

[6] Jerry Hobbs. Ontological promiscuity. In *Proc 23rd Meeting of the ACL*, pages 61–69, NJ, 1985. ACL.

[7] R. A. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12:121–146.

[8] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proc Knowledge Representation '96 (KR'96)*, pages 2–13, 1996.

[9] Benjamin Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, Sept 1994.

[10] J. W. Rickel and B. W. Porter. Automated modeling of complex systems to answer prediction questions. *Artificial Intelligence*, 1997.

# Index