FaCT Reference Manual
Version 1.6

Ian Horrocks
horrocks@cs.man.ac.uk

Department of Computer Science
University of Manchester
Manchester M13 9PL
United Kingdom

August, 1998

Contents

1 Introduction

FaCT is a prototype description logic knowledge representation system which uses an optimised tableaux subsumption algorithm to provide complete inference for a relatively expressive concept description language. In particular, FaCT supports transitively closed roles, a role/attribute hierarchy and general concept inclusion axioms (implications of the form $C \Rightarrow D$ where C and D are arbitrary concept descriptions).

FaCT is primarily intended as a tool for conceptual schema design and ontological engineering and thus provides only concept based reasoning services: there is a TBox but no ABox. The interface to the TBox is designed to be broadly compatible with that of the Kris system [BH91].

The correspondence between $\mathcal{ALC}$ and the propositional modal logic $\mathbf{K_{(m)}}$ means that FaCT can also be used as a highly efficient decision procedure for both $\mathbf{K_{(m)}}$ and $\mathbf{K4}$ [GS96, HS97].

1.1 Obtaining FaCT

FaCT can be obtained from the following world wide web site:

http://www.cs.man.ac.uk/mig/people/horrocks/FaCT

After downloading the file `FaCT.tar.gz` use the command:

```
gunzip FaCT.tar.gz ; tar xvf FaCT.tar
```

to create a directory `FaCT` containing the distribution files and directories.

To install FaCT follow the instructions in the file `FaCT/README`.

FaCT has been written in Common LISP and has been tested with Allegro Common Lisp, Harlequin LispWorks and Gnu Common Lisp.

2 Concept Descriptions

FaCT uses the same list based concept description syntax as the Kris system. If CN is the name of a defined concept, *R* is the name of a defined role, *A* is the name of a defined attribute and $C, C_1, \ldots, C_n$ are concept descriptions, then the following are also valid concept descriptions:

TOP
BOTTOM
CN
(and $C_1 \ldots C_n$)
(or $C_1 \ldots C_n$)
(not C)
(some *R* C)
(all *R* C)
(some *A* C)
(all *A* C)

The correspondence between this form and the standard infix notation is shown in Table 1.

FaCT syntax	Standard notation
`*TOP*`	$\top$
`*BOTTOM*`	$\bot$
(`and` $C_1 \ldots C_n$)	$C_1 \sqcap \ldots \sqcap C_n$
(`or` $C_1 \ldots C_n$)	$C_1 \sqcup \ldots \sqcup C_n$
(`not` C)	$\neg C$
(`some` R C)	$\exists R.C$
(`all` R C)	$\forall R.C$

Table 1: FaCT concept expressions

3 Function and Macro Interface

This section describes the built-in concepts, macros and functions which provide the user interface to the TBox.

3.1 Built-in Concepts

`*TOP*` *concept*

Description: The name of the top concept ($\top$).

Remarks: Every concept in the TBox is subsumed by `*TOP*`.

`*BOTTOM*` *concept*

Description: The name of the bottom concept ($\bot$).

Remarks: Every concept in the TBox subsumes `*BOTTOM*`. Note that incoherent concepts become synonyms for `*BOTTOM*`.

3.2 Knowledge Base Management

init-tkb *function*

Description: Initialises the TBox.

Syntax: (**init-tkb**)

Remarks: All user defined concepts, roles and implications are deleted from the TBox leaving only `*TOP*` and `*BOTTOM*`.

Examples: `(init-tkb)`

load-tkb *function*

Description: Loads a TBox from a file.

Syntax: (**load-tkb** *name* `&key` (*verbose* `T`) (*overwrite* `nil`))

Arguments:
- *name* - Name of the TBox file.
- *verbose* - Keyword which, if non-`nil`, causes the classifier to print symbols indicating the progress of the load operation: `P` for each primitive concept, `C` for each non-primitive concept, `R` for each role, `R+` for each transitive role, `A` for each attribute and `I` for each implication. If omitted, *verbose* defaults to `T`.
- *overwrite* - Keyword which, if non-`nil`, causes the classifier to clears the existing TBox from memory (by performing an **init-tkb**) before loading the new TBox. If omitted, *overwrite* defaults to `nil`.

Return: `T` if the TBox is successfully loaded; `nil` otherwise.

Examples: `(load-tkb 'demo.kb :verbose T)`

3.3 TBox Definitions

defprimconcept *macro*

Description: Defines a primitive concept.

Syntax: (**defprimconcept** *name* `&optional` (*description* `*TOP*`))

Arguments:
- *name* - Name of the new concept.
- *description* - Optional description of the new concept. If omitted it defaults to `*TOP*`.

Return: A concept structure `c[`*name*`]` is returned.

Remarks: The new concept is not classified until a call is made to **classify-tkb**. It is an error if a concept *name* has already been defined.

Examples:

```
(defprimconcept ANIMAL)
(defprimconcept MALE)
(defprimconcept FEMALE)
(defprimconcept BIPED)
(defprimconcept HUMAN (and ANIMAL BIPED))
```

defprimconcept-f *function*

Description: Functional equivalent of **defprimconcept**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(defprimconcept-f 'ANIMAL)
(defprimconcept-f 'MALE)
(defprimconcept-f 'FEMALE)
(defprimconcept-f 'BIPED)
(defprimconcept-f 'HUMAN '(and ANIMAL BIPED))
```

defconcept *macro*

Description: Defines a non-primitive concept.

Syntax: (**defconcept** *name description*)

Arguments: *name* - Name of the new concept.

description - Description of the new concept.

Return: A concept structure `c[`*name*`]` is returned.

Remarks: The new concept is not classified until a call is made to **classify-tkb**. It is an error if a concept *name* has already been defined. Note that in contrast to **defprimconcept** it is also an error if *description* is omitted.

Examples:
```
(defconcept MAN (and MALE HUMAN))
(defconcept WOMAN (and FEMALE HUMAN))
```

defconcept-f *function*

Description: Functional equivalent of **defconcept**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(defconcept-f 'MAN '(and MALE HUMAN))
(defconcept-f 'WOMAN '(and FEMALE HUMAN))
```

defprimrole *macro*

Description: Defines a primitive role.

Syntax: (**defprimrole** *name* `&key` (*supers* `nil`) (*transitive* `nil`))

Arguments: *name* - Name of the new primitive role.

supers - Keyword list of super-roles. If omitted it defaults to `nil`.

transitive - Keyword which, if non-`nil`, makes the role transitive. If omitted it defaults to `nil`.

Return: A role structure `r[`*name*`]` is returned.

Remarks: It is an error if a role or attribute *name* has already been defined.

Examples:
```
(defprimrole Relation :transitive T)
(defprimrole Close-relation :supers (Relation))
(defprimrole Ancestor :supers (Relation) :transitive T)
(defprimrole Parent :supers (Close-relation Ancestor))
```

defprimrole-f *function*

Description: Functional equivalent of **defprimrole**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(defprimrole-f 'Relation :transitive T)
(defprimrole-f 'Close-relation :supers '(Relation))
(defprimrole-f 'Ancestor :supers '(Relation) :transitive T)
(defprimrole-f 'Parent :supers '(Close-relation Ancestor))
```

defprimattribute *macro*

Description: Defines a primitive attribute (functional role).

Syntax: (**defprimattribute** *name* `&key` (*supers* `nil`))

Arguments: *name* - Name of the new primitive attribute.

supers - Keyword list of super-attributes or super-roles. If omitted it defaults to `nil`.

Return: A role structure `r[`*name*`]` is returned.

Remarks: It is an error if a role or attribute *name* has already been defined. Note that unlike roles, attributes cannot be transitive.

Examples:
```
(defprimattribute Best-friend)
(defprimattribute Father :supers (Parent))
(defprimattribute Mother :supers (Parent))
```

defprimattribute-f *function*

Description: Functional equivalent of **defprimattribute**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(defprimattribute-f 'Best-friend)
(defprimattribute-f 'Father :supers '(Parent))
(defprimattribute-f 'Mother :supers '(Parent))
```

implies *macro*

Description: An implication/subsumption axiom between two concept descriptions.

Syntax: (**implies** *antecedent consequent*)

Arguments: *antecedent* - Description of the antecedent concept.

consequent - Description of the consequent concept.

Return: *antecedent* is returned.

Remarks: Asserts that *antecedent* implies *consequent* (*antecedent* $\Rightarrow$ *consequent*) or, equivalently that the *antecedent* is subsumed by *consequent* (*antecedent* $\sqsubseteq$ *consequent*). Note that adding implications *after* the TBox has been classified may cause strange and unpredictable results.

Examples:
```
(implies (and POLYGON ((some Angles 3))) (some Sides 3))
```

implies-f *function*

Description: Functional equivalent of **implies**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(implies-f '(and POLYGON ((some Angles 3))) '(some Sides 3))
```

disjoint *macro*

Description: A disjointness axiom between concept descriptions.

Syntax: (**disjoint** *description-1 description-n*)

Arguments: *description-i*- A concept description.

Remarks: Asserts that the extensions of *description-1*,...,*description-n* are disjoint.

Examples:
```
(disjoint MALE FEMALE)
(disjoint CAT DOG RABBIT HAMSTER)
```

disjoint-f *function*

Description: Functional equivalent of **disjoint**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(disjoint-f 'MALE 'FEMALE)
(disjoint 'CAT 'DOG 'RABBIT 'HAMSTER)
```

3.4 TBox Inferences

classify-tkb *function*

Description: Classifies the TBox.

Syntax: (**classify-tkb** `&key` (*mode* `:nothing`))

Arguments: *mode* - Keyword which controls the output from the classifier:

- `:nothing` - None;
- `:stars` - A symbol for each concept classified: `P` for a primitive concept, `C` for a non-primitive concept and `S` for a synonym;
- `:names` - The name of each concept classified followed by `-P` for a primitive concept, `-C` for an non-primitive concept and `-S` for a synonym;
- `:count` - The number of subsumption and satisfiability tests.

Warnings are always output regardless of the setting of *mode*. If *mode* is either `:stars` or `:names` then symbols are also output during the pre-processing of roles (`r` for each role `a` for each attribute), implications (`p` for each implication absorbed into a primitive concept, `i` for each non-absorbed implication) and concept terms (`c` for each concept term). If omitted, *mode* defaults to `:nothing`.

Return: Returns `T` if the current TBox is coherent (i.e., the concept `*TOP*` is satisfiable), `nil` otherwise.

Remarks: If any new implications have been added to the TBox (**implies** or **f-implies**) since the last TBox classification (**classify-tkb**), all concepts will be reclassified; otherwise classification will be incremental.

Examples:
```
(classify-tkb :mode :stars)
```

direct-supers *function*

Description: Finds the direct super-concepts of a classified concept.

Syntax: (**direct-supers** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of the direct subsumers (super-concepts) of the concept *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

Examples: `(direct-supers 'MAN)` ⇒ `(c[HUMAN] c[MALE])`

---

all-supers *function*

Description: Finds all the super-concepts of a classified concept.

Syntax: (**all-supers** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of all the subsumers (super-concepts) of the concept *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

Examples: `(all-supers 'MAN)` ⇒ `(c[ANIMAL] c[HUMAN] c[BIPED] c[MALE] c[*TOP*])`

---

direct-subs *function*

Description: Finds the direct sub-concepts of a classified concept.

Syntax: (**direct-subs** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of the direct subsumees (sub-concepts) of the concept *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

Examples: `(direct-subs 'MALE)` ⇒ `(c[MAN])`

---

all-subs *function*

Description: Finds all the direct sub-concepts of a classified concept.

Syntax: (**all-subs** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of all the subsumees (sub-concepts) of the concept *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

Examples: `(all-subs 'MALE)` ⇒ `(c[MAN] c[*BOTTOM*])`

---

equivalences *function*

Description: Finds those concepts which are equivalent to a classified concept.

Syntax: (**equivalences** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of all the concepts which are equivalent to (synonyms for) *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

---

satisfiable *function*

Description: Tests if a concept description is satisfiable.

Syntax: (**satisfiable** *description*)

Arguments: *description* - A concept description.

Return: Returns `T` if *description* is satisfiable w.r.t. the current TBox, `nil` otherwise.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description* must be quoted.

Examples: `(satisfiable '(and MALE FEMALE))` ⇒ `nil`
`(satisfiable '(and MALE ANIMAL))` ⇒ `T`

---

subsumes *function*

Description: Tests if one concept description subsumes another.

Syntax: (**subsumes** *description-1 description-2*)

Arguments: *description-1* - A concept description.
description-2 - A concept description.

Return: Returns `T` if *description-1* subsumes *description-2* w.r.t. the current TBox, `nil` otherwise.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description-1* and *description-2* must be quoted.

Examples: `(subsumes '(and MALE ANIMAL) 'MAN)` ⇒ `T`

---

equivalent-concepts *function*

Description: Tests if two concept descriptions are equivalent.

Syntax: (**equivalent-concepts** *description-1 description-2*)

Arguments: *description-1* - A concept description.

description-2 - A concept description.

Return: Returns `T` if *description-1* is equivalent to *description-2* w.r.t. the current TBox, `nil` otherwise.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description-1* and *description-2* must be quoted.

Examples: `(equivalent-concepts '(and MALE HUMAN) 'MAN)` ⇒ `T`
`(equivalent-concepts 'HUMAN '(and ANIMAL BIPED))` ⇒ `nil`

---

disjoint-concepts *function*

Description: Tests if two concept descriptions are disjoint.

Syntax: (**disjoint-concepts** *description-1 description-2*)

Arguments: *description-1* - A concept description.

description-2 - A concept description.

Return: Returns `T` if *description-1* is disjoint from *description-2* w.r.t. the current TBox, `nil` otherwise.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description-1* and *description-2* must be quoted.

Examples: `(disjoint-concepts 'MALE 'FEMALE)` ⇒ `T`

---

classify-concept *function*

Description: Finds where a concept description would classify without adding it to the TBox.

Syntax: (**classify-concept** *description*)

Arguments: *description* - A concept description.

Return: Returns three values: a list of all the direct-supers of *description*; a list of all the direct-subs of *description*; a list of all the concepts which are equivalent to *description*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description* must be quoted.

Examples: `(classify-concept '(and MALE ANIMAL))` ⇒ `(c[MALE] c[ANIMAL]) (c[MAN]) nil`

---

add-concept *macro*

Description: Defines a new concept and classifies the TBox using **classify-tkb**.

Syntax: (**add-concept** *name description* `&key` (*primitive* `nil`))

Arguments: *name* - Name of the new concept.

description - Description of the new concept.

primitive - Keyword which, if non-`nil`, makes the concept primitive. If omitted it defaults to `nil`.

Return: A concept structure `c[`*name*`]` is returned.

Remarks: It is an error if a concept *name* has already been defined.

Examples: `(add-concept VEGETABLE *TOP* :primitive T)`
`(add-concept WOMAN (and FEMALE HUMAN))`

---

add-concept-f *function*

Description: Functional equivalent of **add-concept**.

Remarks: Note that all arguments have to be quoted.

Examples: `(add-concept-f 'VEGETABLE '*TOP* :primitive T)`
`(add-concept-f 'WOMAN '(and FEMALE HUMAN))`

3.5 TBox Queries

---

get-concept *function*

Description: Retrieves a concept from the TBox.

Syntax: (**get-concept** *name*)

Arguments: *name* - Concept name (a LISP atom).

Return: A concept structure `c[`*name*`]` if a concept *name* is defined in the TBox; `nil` otherwise.

get-role *function*

Description: Retrieves a role or attribute from the TBox.

Syntax: (**get-role** *name*)

Arguments: *name* - Role or attribute name (a LISP atom).

Return: A role structure `r[`*name*`]` if a role or attribute *name* is defined in the TBox; `nil` otherwise.

get-all-concepts *function*

Description: Retrieves all concepts from the TBox.

Syntax: (**get-all-concepts**)

Return: A list of all the concepts defined in the TBox.

get-all-roles *function*

Description: Retrieves all roles and attributes from the TBox.

Syntax: (**get-all-roles**)

Return: A list of all the roles and attributes defined in the TBox.

classified-tkb? *function*

Description: Tests if the TBox is classified.

Syntax: (**classified-tkb?**)

Return: `T` if the TBox is classified; `nil` otherwise.

what-is? *function*

Description: Determines the type of a concept, or role structure.

Syntax: (**what-is?** *structure*)

Arguments: *structure* - Concept or role structure.

Return: The type of structure, one of `CONCEPT`, `PRIMITIVE`, `ROLE` or `FEATURE`.

is-primitive? *function*

Description: Determines if a structure is a primitive concept (of type `PRIMITIVE`).

Syntax: (**is-primitive?** *structure*)

Arguments: *structure* - Concept structure.

Return: `T` if *structure* is a primitive concept (of type `PRIMITIVE`); `nil` otherwise.

is-concept? *function*

Description: Determines if a structure is a non-primitive concept (of type `CONCEPT`).

Syntax: (**is-concept?** *structure*)

Arguments: *structure* - Concept structure.

Return: `T` if *structure* is a non-primitive concept (of type `CONCEPT`); `nil` otherwise.

is-role? *function*

Description: Determines if a structure is a role (of type `ROLE`).

Syntax: (**is-role?** *structure*)

Arguments: *structure* - Role structure.

Return: `T` if *structure* is a role (of type `ROLE`); `nil` otherwise.

is-feature? *function*

Description: Determines if a structure is an attribute (of type `FEATURE`).

Syntax: (**is-feature?** *structure*)

Arguments: *structure* - Role structure.

Return: `T` if *structure* is an attribute (of type `FEATURE`); `nil` otherwise.

name *function*

Description: Retrieves the name of a concept or role structure.

Syntax: (**name** *structure*)

Arguments: *structure* - Concept or role structure.

Return: The name of *structure* if it is a concept or role structure; `nil` otherwise.

---

description *function*

Description: Retrieves the description of a concept or role structure.

Syntax: (**description** *structure*)

Arguments: *structure* - Concept or role structure.

Return: The description of *structure* if it is a concept or role structure; `nil` otherwise. The format of the description depends on the type of structure:

- `PRIMITIVE` - The *description* given in the **defprimconcept** or **defprimconcept-f** definition, possibly extended by absorption of **implies** axioms.
- `CONCEPT` - The *description* given in the **defconcept** or **defconcept-f** definition.
- `ROLE` - A list consisting of the role's name, the keyword `:supers` followed by a list of the role's *supers* given in the **defprimrole** or **defprimrole-f** definition and the keyword `:transitive` followed by `T` if the role is transitive (the last 2 items will be omitted if their values are `nil`).
- `FEATURE` - A list consisting of the attribute's name and the keyword `:supers` followed by a list of the attribute's *supers* given in the **defprimattribute** or **defprimattribute-f** definition (the last item will be omitted if its value is `nil`).

4 Controlling FaCT's Behavior

This section describes functions and macros which allow the classifiers features to be customised and which control performance profiling.

---

set-features *function*

Description: Enables classifier features.

Syntax: (**set-features** `&rest` *features*)

Arguments: *features* - Zero or more keywords. The feature associated with each supplied keyword is enabled. The available keywords are as follows:


```
:transitivity :concept-eqn :subset-s-equivalent
:backjumping :obvious-subs :top-level-caching
:full-caching :blocking :taxonomic-encoding
:gci-absorption :cyclical-definitions
:auto-configure :moms-heuristic :prefer-pos-lits
:minimise-clashes :auto-install-primitives
:auto-install-transitive
```

By default all features are enabled except:

`:moms-heuristic`, `:auto-install-primitives`
`:auto-install-transitive`

Return: A list of all enabled features.

Remarks: See source code for details of the effect of each feature.

Examples:
```
(set-features)
(set-features :transitivity :backjumping)
```

---

reset-features *function*

Description: Disables classifier features.

Syntax: (**reset-features** `&rest` *features*)

Arguments: *features* - Zero or more keywords. The feature associated with each supplied keyword is disabled.

Return: A list of all enabled features.

Remarks: See **set-features** for a list of features. See source code for details of the effect of each feature.

Examples: `(reset-features :transitivity :backjumping)`

---

features *function*

Description: Prints information about feature settings.

Syntax: (**features** `&optional` (*stream* `T`))

Arguments: *stream* - Optional output stream; default is `T` (`*terminal-io*`).

Examples: `(features)`

---

set-profiling *function*

Description: Controls performance profiling.

Syntax: (**set-profiling** `&key` (*level* 1) (*file* "profile.out"))

Arguments: *level* - Keyword which controls the amount of profiling data which is output:

- 0 - Disables profiling.
- 1 - Outputs profiling data for each concept classification.
- 2 - Outputs profiling data for each subsumption test.
- 3 - Outputs profiling data for each satisfiability test.

If omitted, *level* defaults to 1.

file - Keyword specifying the name of a file to which the profiling data is to be written. If explicitly `nil` it is written to the `*terminal-io*` stream. If omitted, *file* defaults to "profile.out".

Remarks: Note that profiling can generate a large amount of data, particularly if `level` is >1. For each satisfiability test the profiler outputs the number of backtracks, the maximum model size, the maximum model depth, the number of cache accesses, the number of cache hits, the CPU time used, the result (`T` or `nil`) and whether blocking was triggered (`T` or `nil`). See source code for more details.

Examples:
```
(set-profiling)
(set-profiling :level 2 :file nil)
```

---

reset-profiling *function*

Description: Disables performance profiling.

Syntax: (**reset-profiling**)

Remarks: Equivalent to `(set-profiling :level 0)`.

Examples:
```
(reset-profiling)
```

---

set-verbosity *function*

Description: Controls the verbosity of the classifier.

Syntax: (**set-verbosity** `&rest` *features*)

Arguments: *features* - Zero or more keywords. The verbosity feature associated with each supplied keyword is enabled. The available keywords are as follows:

`:warnings :notes :synonyms :reclassifying :features :rc-counts :test-counts :cache-counts :classify-1 :classify-2`

By default, the of enabled verbosity features are:

`:warnings :synonyms :reclassifying :rc-counts :test-counts :cache-counts`

Return: A list of all enabled verbosity features.

Remarks: The verbosity setting is temporarily overridden by the *verbose* argument to **load-tkb** and the *mode* argument to **classify-tkb**. See source code for details of the effect of each verbosity feature.

Examples:
```
(set-verbosity)
(set-verbosity :notes :classify-2)
```

---

reset-verbosity *function*

Description: Reduces the verbosity of the classifier.

Syntax: (**reset-verbosity** `&rest` *features*)

Arguments: *features* - Zero or more keywords. The feature associated with each supplied keyword is disabled. If no keywords are supplied, all verbosity features are disabled.

Return: A list of all enabled verbosity features.

Remarks: See **set-features** for a list of verbosity features. See source code for details of the effect of each feature.

Examples:
```
(reset-verbosity)
(reset-verbosity :notes :classify-2)
```

5 Modal Logic Theorem Proving

FaCT can also be used as a decision procedure for the propositional modal logics $\mathbf{K_{(m)}}$ and **K4**. Although the functions and macros described in Section 3 could be used for this purpouse, FaCT provides a more convenient functional interface for performing single satisfiability tests on $\mathbf{K_{(m)}}$ and **K4** formulae encoded as an $\mathcal{ALC}$ concept description.

---

alc-concept-coherent *function*

Description: Tests the satisfiability of an $\mathcal{ALC}$ concept.

Syntax: (**alc-concept-coherent** *description* `&key` (*k4* `nil`))

Arguments: *description* - Concept description.

k4 - Keyword which, if non-`nil`, causes *description* to be treated as a **K4** formula (all roles are transitive). If omitted it defaults to `nil`.

Return: Returns four values: `T` if *description* is satisfiable, `nil` otherwise; run time (seconds) *excluding* concept encoding; number of backtracks; maximum model size (nodes).

Remarks: All user defined concepts, roles and implications are deleted from the TBox (using **init-tkb**); some or all of the concepts and roles occuring in *description* are then automatically installed in the TBox. The verbosity setting is temporarily overridden and set to `nil`. Profiling is temporarily disabled.

Examples:

```
(alc-concept-coherent '(and (some R (some R C1))
                            (all R (not C1))))
```

```
(alc-concept-coherent '(and (some R (some R C1))
                            (all R (not C1))) :k4 T)
```


References

[BH91] F. Baader and B. Hollunder. KRIS: Knowledge representation and inference system. *SIGART Bulletin*, 2(3):8–14, 1991.

[GS96] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for $\mathcal{ALC}$. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *Principals of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, pages 304–314. Morgan Kaufmann, November 1996.

[HS97] U. Hustadt and R. A. Schmidt. On evaluating decision procedures for modal logic. Technical Report MPI-I-97-2-003, Max-Planck-Institut Für Informatik, February 1997.