

SHADE Deductive-DB/Content-Based-Routing Agent

- [General Description](#)
- [How to Startup](#)
- [How to Kill a Remote Agent](#)
- [Recognized KQML Performatives](#)
- [Supported Builtin KIF](#)
- [Caveats](#)
- [Scenarios of Usage](#)

General Description

A key facilitation service in agent-based architectures is content-based routing. In content-based routing, messages sent by an agent are routed based on interests asserted by other agents rather according to specific, prearranged address. This allows agents, who don't know of each other's existence, to establish communication and share information. As the network of agents becomes very large and dynamic, absence of a priori knowledge will become the norm rather than the exception.

To realize content-based routing, agents must be able to express their interests in terms of a general subscription. In the simplest case, a subscription looks like a syntactic pattern that must be unified against the message. More generally, it looks like an arbitrarily complex first-order logic condition that must be evaluated for satisfiability with respect to an ontology. An agent subscribes to information by posing a persistent query within the vocabulary of the common ontology. When relevant information is published by some other agent, it is picked up by the routing agent and forwarded on to the requesting agent as though told directly by the publisher.

An initial KQML/KIF-literate content-based router and deductive database agent has been implemented. The router has been combined with a deductive database agent since both must understand KIF and KQML and, depending on the complexity of the routing desired, arbitrary inference may be required. This has the added benefit of making available a deductive database agent.

The front-end of the router is implemented in C++, and the back-end inferencing of this agent is performed by a public-domain prolog system (Stony Brook Prolog). The 'C' source code of SB-Prolog has been modified to allow its encapsulation as an embeddable C++ object. A KIF parser developed on Lockheed's Knowledge Centered Design IR was enhanced to recognize a greater subset of KIF and to provide translation from KQML/KIF into an equivalent interpretable Prolog representation. The agent also employs the KQML API.

In addition to supporting standard performatives (such as ask, tell, evaluate, stream), the agent supports a subscribe/monitor of "ask-about" and "stream-about". The subscribe/monitor performative is a request for change only notification. The ask-about/stream-about performatives return all sentences known about an object. These performatives depend on the sentences being simple database tables (i.e. incompletely defined relations and not contained in an implication). Even though it makes sense to build in monitor/subscribe for completely defined relations/functions and for implications, this has not been done since it will require a built-in truth-maintenance component over forward chaining rules.

Special care has been taken to build into the agent support for many of the special KIF functions and relations such as those pertaining to lists, sets, numbers, concretions of relations and functions, and metaknowledge representation and reasoning. Currently the agent can support inference over built-in relations/functions and base-table relations/functions. Functional evaluation of terms is also currently supported. For example,

```
(item (first (listof 1 2 3))
      (butlast (listof 3 4 1 2)))
```

which is equivalent to

```
(item 1 (listof 3 4 1))
```

which is a true sentence. The remaining work to be accomplished lies in supporting complete definitions of derived relations and functions as well as supporting implication.

An open issue in content-based routing is the extent to which the content-based router must rely on context-specific information to evaluate the notification criteria. If the message and the supporting ontology contain all information needed to evaluate the interest, as in Figure~2, then the scheme works fine. If, however, the content-based router needs to remember context-specific information (i.e., the values of ports over time in some simulation), then the approach may become infeasible due to space and efficiency concerns.

Within the current content-based router implementation, informational updates are examined by the router for relevance to other agents but are not incorporated indiscriminately into the routing agent's local model. KQML provides directives to discriminate between informational updates which can be forgotten (e.g. "tell" in Figure~2), and informational updates which are also requests to remember the message within the recipient's knowledge base (e.g. "insert"). The routing agent makes the assumption that any context-specific information which it needs to remember has been provided to it via the "insert" directive.

How to Startup

There is currently no install tape.

First, make sure that you've sourced the file `$$SHADE/sbprolog/v3/sbp_v3.1/INIT` (at Lockheed, `$$SHADE` is `/aic/shade`).

There are two executables: "test_api" and "deductive_db_agent". Upon starting these up, you will be queried for the host where the mbus server is running, the port which the server is accepting client requests, and finally the name of your agent. You can invoke as many of each as you want to create a family of interoperating agents.

The "test_api" code provides a convenient way to send and receive KQML messages. ALL THE EXAMPLES WHICH FOLLOW DO NOT CONTAIN `:sender` and `:receiver` parameters ONLY BECAUSE THE test_api CODE ADDS THAT INFO. These parameters are necessary.

The "deductive_db_agent" is the actual database agent and content-based router. Most of its behavior is manifested via KQML messages (seen only via the test_api), but it also spits out nonsensical debugging data.

How to Kill a remote agent in KQML, from another agent

A convenient way to kill processes:

```
(achieve :sender :receiver
         :language kqml
         :content (unregister :name ))
```

Recognized KQML Performatives

The different KQML messages that the deductive database agent can process are listed below, grouped according to their general functionality and annotated with comments.

Upon receiving these, the agent will not update it's VKB, but instead will simply do content-based routing. Untells of definitions are not recommended (this is treated as an abolish, which effectively wipes out all known tuples).

- tell
- untell

Agent will reflect update in it VKB, and will do content-based routing.

- insert
- delete
- monitor
- subscribe

Currently the agent only handles denies of monitor and subscribe.

- deny
- ask-one
- ask-all
- ask-about
- ask-if
- stream-all
- stream-about
- evaluate
- reply

Need to provide a performative which indicates "no more answers". Sorry is overloaded, since impossible to distinguish between cannot do and no answers.

- sorry

Killing agents:

```
(achieve :language kqml :content (unregister :name agent1))
kills agent1

(unachieve :language kqml :content (register :name agent1))
kills agent1
```

Supported Builtin KIF

Builtin Relations:

- true
- integer
- 'real-number'
- number
- natural
- '/='
- =
- <
- >
- >=
- =<
- zero
- positive
- negative
- 'odd-integer'
- 'even-integer'
- list
- null
- single
- double
- triple
- item
- empty
- set
- member
- subset
- disjoint
- relation
- function
- //concretion
- holds

Builtin Functions

- '*'
- '+'
- '-'
- '/'
- '1+'
- '1-'
- abs
- max
- min
- if
- if
- name
- denotation
- first
- rest
- last
- butlast
- cons
- append
- reverse
- adjoin
- remove
- length

- nth
- nthrest
- subst
- union
- difference
- intersection
- setofall
- //relation/function concretion
- value
- apply
- map

Term Ops ("cond" and "the" are not handled). Also, backquote and comma notation are not supported, have to use listof.

- quote
- if
- setof
- listof

Sentence Ops

- not
- and
- or
- <= , =>
- exists
- forall

Either definition is completely specified or entirely incompletely specified.

- defobject
- defrelation
- deffunction
- ':='

Caveats

Ontology parameters are virtually ignored. They are passed along, but they do not affect inference in any manner.

It is not fair to say that it is a true KIF interpreter. There are a lot of embedded closed world assumptions. It tries to be three-valued (T,F,don't know), but does not always succeed. This raises the issue of how best to characterize this. Many real-world agents are closed-world. KIF SYNTAX is a reasonable neutral format. How should these agents specify, "I can handle KIF syntax, but I make a closed world assumption"? The answer cannot be, "Gee, well axiomatize your unique-names assumptions, etc.

Tells/untells are not persistent. You need to use insert/delete to update the agents VKB. It would be nice to update the code so the deductive db agent could be told what it should be interested in, and these are persistently maintained. This should be easily doable. This is just a special case of content-based routing where the interested party is the same agent.

Be careful about the use of existentially quantified variables. Many times, an existentially quantified variable has local scope, allowing several non-unifiable

instances of the same variable within a sentence.

```
Ex:
  (and (man ?y)
        (exists (?x) (wife-of ?y ?x))
        (exists (?x) (girl-friend ?y ?x)))
```

The variables ?x should not unify. However, in the KIF interpreter, the variable bindings within the (exists ..) have global scope, and therefore would unify. This is a bug, but it is difficult to fix (need a copy-term mechanism, and short of asserting to and then retracting from the db, there is no efficient mechanism available). Thus when using the interpreter, make sure you introduce new variables to prevent unintended unifications.

Function tuples, when asserted via a tell, must be in sentence form:

```
Ex:
  (= (f 1 2 4) ?value)
```

Within the :content of a query, defined functions can be treated as relations

```
Ex:
  (and (f 1 2 3 4 ?value)
        ... )
```

HOWEVER, builtin functions are not treated as relations since many of them are unsafe or cannot be handled:

```
Ex:
  (true (+ 1 2 3)) is not handled
  (true (+ 1 ?x 3)) is not handled - cannot do CLP
  (true (+ ?x ?y 3)) is not safe
```

Did not implement all the exotic boundedness axiomatization stuff. The sentence (set ?v) is true if ?v evaluates to (setof @args). If ?v does not evaluate to (setof @args), then agent may incorrectly return bottom (because of the hidden closed-world assumptions).

Ask-about/stream-about only work on atomic objects, and only if defined via an incompletely defined.

```
(defobject obj)
```

```
Ex: cannot (ask-about :language kif :content foo)
      when (defobject foo := (listof a b c)) is true.
```

When a sentence is incorporated into the agent's VKB, all unrecognized words are assumed to be objects and are handled as though there existed a previously asserted (defobject ..). A cross-referencing web is created to efficiently answer ask-about/stream-about.

It is a good idea to introduce definitions first (i.e. load your relevance theory). Don't do a definition after the constant/object in question has already been seen.

Doesn't support untells of complete definitions. Untells of incompletely defined definitions or implications are treated as a purge.

For setofall to work properly, guarantee that all existentially quantified variables are moved to outermost level of the sentence.

```
Ex:
  (setofall (listof ?x) (exists (?y) (and (mother ?x ?y) ...)))
```

vs.

```
(setofall (listof ?x) (and (exists (?y) (mother ?x ?y)) ..))
```

These are semantically the same thing, but they will be treated differently. I don't wade through the conditions of a setofall to locate all existentially quantified vars – I assume they are specified at the outermost layer.

Note: existentially quantifying variables means that you don't care about the value of those variables outside the scope of the setofall. If there are free vars within the setofall, then the setofall condition can become a backtracking point. This is correct behavior, which most non logic programming folks are unaware of.

Monitor assumes that :aspect is an equality, (= (f @s) ?v), or is a defined relation, which is also included in the :content.

Scenarios of Usage

```
/***** Startup an agent *****/

% test_api
Specify Host: fermanagh
Specify MBUS Port: 2390
Specify Your Agent Name: jim
calling KBegin

/***** Ask queries using builtin KIF relations/functions*****/

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(evaluate :language kif
  :content (if (item 5 (append (listof 1 2 3) (listof 4 5 6)))
    (intersection (setof 1 2 3) (setof 3 4 5))
    nil)
  :reply-with yowser).

(r)ead,(w)rite, or (q)uit: r
got from bc msg:
(reply :sender bc :receiver jim :in-reply-to yowser
  :language kif
  :content (setof 3))

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(ask-if :language kif
  :content (true (listof (quote >)
    (quote (length (listof 1 2 3)))
    (quote 2)))
  :reply-with hello).

(r)ead,(w)rite, or (q)uit: r
got from bc msg:
(reply :sender bc :receiver jim :in-reply-to hello :language kif :content true)

/***** Provide some definitions *****/
// If you provide a :reply-with in insert/delete/tell/untell, an
// acknowledgement will come back

// Use of object evaluation and meta-knowledge handling:
// I've since learned that is not quite right. My code greedily attempts
// to evaluate within a quote, but the evaluation should be delayed
```

```

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(insert :language kif
    :content (defobject a := (setof 1 2 3))).

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(ask-if :language kif
    :content (= (quote (f a b c))
        (listof (quote f) (quote (setof 1 2 3)) (quote b) (quote c)))
    :reply-with yow).

(r)ead,(w)rite, or (q)uit: r
got from bc msg:
(reply :sender bc :receiver jim :in-reply-to yow :language kif :content true)

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(insert :language kif :content (defrelation mother (?x ?y))).

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(insert :language kif
    :content (defrelation mother (?x ?y))).

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(insert :language kif
    :content (defrelation parent (?x ?y) :=
        (or (mother ?x ?y) (father ?x ?y)))).

/**** Assert some tuples *****/
/**** Inserts/Deletes of completely defined relation disallowed. **
**** This is the view update problem in db theory */

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(insert :language kif
    :content (and (father f1 c) (father f2 d) (father f3 e))).

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(insert :language kif
    :content (and (mother b c) (mother c d) (mother d e))).

/****Issue Some Queries over the defined relations *****/

/****The reply for a stream-xxx comes back as tells if no :aspect provided.
****Otherwise the answer comes back as a series of reply's. An aspect allows
****arbitrary patterns to be constructed, so answering with tell's could
****lead to unintended inconsistent assertions
****/

(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(stream-all :language kif :content (father ?x ?y) :reply-with the-answer).

(r)ead,(w)rite, or (q)uit: r

```

```

got from bc msg:
  (tell :sender bc :receiver jim :in-reply-to the-answer :language kif
    :content (father f1 c))

(read,(w)rite, or (q)uit: r
got from bc msg:
(tell :sender bc :receiver jim :in-reply-to the-answer :language kif
  :content (father f2 d))

(read,(w)rite, or (q)uit: r
got from bc msg:
(tell :sender bc :receiver jim :in-reply-to the-answer :language kif
  :content (father f3 e))

(read,(w)rite, or (q)uit: r
got from bc msg:
(sorry :sender bc :receiver jim :in-reply-to the-answer :language kif)

(read,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(ask-all :language kif :aspect (listof ?x ?y)
  :content (mother ?x ?y)
  :reply-with yowser).

(read,(w)rite, or (q)uit: r
got from bc msg:
(reply :sender bc :receiver jim :in-reply-to yowser :language kif
  :content (listof (listof b c) (listof c d) (listof d e)))

(read,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:(stream-about :language kif :content c :reply-with yowser2).

(read,(w)rite, or (q)uit: r
got from bc msg:
(tell :sender bc :receiver jim :in-reply-to yowser2 :language kif
  :content (father f1 c))

(read,(w)rite, or (q)uit: r
got from bc msg:
(tell :sender bc :receiver jim :in-reply-to yowser2 :language kif
  :content (mother b c))

(read,(w)rite, or (q)uit: r
got from bc msg:
(tell :sender bc :receiver jim :in-reply-to yowser2 :language kif
  :content (mother c d))

(read,(w)rite, or (q)uit: r
got from bc msg:
(sorry :sender bc :receiver jim :in-reply-to yowser2 :language kif)

(read,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(ask-all :language kif :aspect ?x
  :content (and (member ?x (setofall ?x2
    (exists (?y ?y2) (and (mother ?y ?x2)
      (father ?y2 ?x2)
      (/= ?y ?y2))))))
  (item ?x (listof a b c d e f)))
  :reply-with yowser3).

(read,(w)rite, or (q)uit: r
got from bc msg:
(reply :sender bc :receiver jim :in-reply-to yowser3 :language kif
  :content (listof c d e))

```

```

// Ask query over derived relation

(read,(write, or (quit): w
Specify Recipient: bc
Specify Message:
(ask-all :language kif :aspect (listof ?x ?y)
          :content (parent ?x ?y) :reply-with the-answer).

(read,(write, or (quit): r
got from bc msg:
(reply :sender bc :receiver jim :in-reply-to the-answer :language kif
:content (listof (listof b c) (listof c d) (listof d e) (listof f1 c)
                (listof f2 d) (listof f3 e)))

/*****Employ Content-Based Routing Functionality*****/

Scenario 1:

Step 1: Introduce a third agent, called mike

% test_api
Specify Host: fermanagh
Specify MBUS Port: 2390
Specify Your Agent Name: mike
calling KBegin

Step 2: Mike posts an interest in all sentences involving the object "foo"

(read,(write, or (quit): w
Specify Recipient: bc
Specify Message:
(subscribe :language kqml :reply-with foo-monitor
          :content (stream-about :language kif :content foo)).

Step 3: Agent bc accepts the request. This is seen from Mike's perspective.
Note: The agent can only monitor :content's which match a defined
sentence schema.

(read,(write, or (quit): r
got from bc msg:
(reply :sender bc :receiver mike :in-reply-to foo-monitor :language kif
:content true)

Step 4: Agent jim assert some new info

(read,(write, or (quit): w
Specify Recipient: bc
Specify Message:
(tell :language kif
      :content (and (father foo a) (father b foo) (father o p))).

Step 5: Agent mike is told of the changes by agent "bc"

(read,(write, or (quit): r
got from bc msg:
(tell :sender jim :receiver mike :in-reply-to foo-monitor :language kif
      :content (father foo a))
(read,(write, or (quit): r
got from bc msg:
(tell :sender jim :receiver mike :in-reply-to foo-monitor :language kif
      :content (father b foo))

/***** Content-Based Routing Functionality (cont'd)*****/

Scenario 2: Demonstration of content-based routing using deductive
matching of messages against notification criteria. The general

```

structure of this type of content-based routing request is

```
(monitor :language kif :aspect <defined-sentence-schema>
  :content (and <defined-sentence-schema>
    ...))
```

Step1: Agent mike posts its interests to agent BC

```
(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(monitor :language kif :aspect (father ?x ?y)
  :content (and (father ?x ?y)
    (exists (?z)
      (and (parent ?z ?y)
        (mother ?z ?y)
        (/= ?z ?x))))
  :reply-with monitor-handle2).
```

Step 2: Agent BC accepts the request

```
(r)ead,(w)rite, or (q)uit: r
got from bc msg:
(reply :sender bc :receiver mike :in-reply-to monitor-handle2 :language kif
  :content true)
```

Step 3: Agent jim asserts something relevant.

```
(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(tell :language kif :content (father f3 e)).
```

Step 4: Agent BC using its relevance theory of definitions and asserted tuples, matches the interests and notifies mike.

```
(r)ead,(w)rite, or (q)uit: r
got from bc msg:
(tell :sender jim :receiver mike :in-reply-to monitor-handle2 :language kif
  :content (father f3 e))
```

/***** Content-Based Routing Functionality (cont'd)*****/

Scenario 3: Demonstration of a request for content-based routing getting turned down. This will occur if the :aspect is not a defined sentence schema (within the service providers relevance theory - which is whatever it has been told to insert into its VKB). Another reason for rejection is that the :content and :aspect fields are not defined as such:

```
(monitor :language kif :aspect <defined-sentence-schema>
  :content (and <defined-sentence-schema>
    ...))
```

Step 1: mike asks for monitoring of an unknown relation, r. To be known, it had to be inserted into the service providers VKB.

```
(r)ead,(w)rite, or (q)uit: w
Specify Recipient: bc
Specify Message:
(monitor :language kif :aspect (r ?x ?y) :content (and (r ?x ?y) (= ?x ?y))).
```

Step2: Agent BC turns down the request

```
(r)ead,(w)rite, or (q)uit: r
got from bc msg: (sorry :sender bc :receiver mike)
```

