KQML Prototype Interface
Final Report
for
Paramax Subcontract No. V20275
Interface Standards for Knowledge Representation Systems

Stuart C. Shapiro and Hans Chalupsky
Department of Computer Science
and Center for Cognitive Science
State University of New York at Buffalo
226 Bell Hall
Buffalo, New York 14260

May 15, 1992

Contents

1 Status and Scope

This report documents an implementation of a KQML (Knowledge Query and Manipulation Language) prototype interface for SNePS-2.1 (the Semantic Network Processing System [Shapiro, 1979; Shapiro and Rapaport, 1987; Shapiro and Rapaport, 1992]). The interface allows two SNePS-2.1 applications to connect to each other via an Internet stream and exchange information by sending and receiving KQML packages and performatives as they are defined in [Shapiro and Chalupsky, 1991]. In this scenario, one application acts as a KQML server (or provider), and the other as a client (or requester) who can query the server about various things. The server maps received KQML requests into appropriate SNePS-2.1 commands to achieve the desired actions, executes them in its SNePS-2.1 knowledge base, and sends appropriate KQML replies back to the client.

All of the performatives defined in [Shapiro and Chalupsky, 1991] with exception of control messages have been implemented. Bidirectional translation of content sentences in SNePSLin (a linear representation of SNePS networks) and SNePSUL (the SNePS User Language) is fully supported. Proper handling of various classes of error exceptions is provided, too. The networking part of the interface is built on top of a TCP/IP package written by Rich Fritzson. An example set of SNePSUL remote commands has been written to show how a SNePS-2.1 application can request information from a KQML server by sending and receiving KQML packages.

2 Overall Design

The main part of the interface is implemented in Common-Lisp as specified in [Steele, 1990]. It uses the Common-Lisp Object System (CLOS) as well as the condition system (both of these parts of the language are not defined in the first edition of the Common-Lisp specification). The networking and server part make use of non-standard extensions such as foreign function calls to establish TCP/IP streams, and asynchronous processes for the servers to be run in. These non-standard extensions are available in Sun (Lucid) Common-Lisp 4.0.1.

The interface is designed to enable a KQML client application to communicate with multiple KQML servers (or providers), as well as allow a KQML server to communicate with multiple KQML clients. Even though an application can communicate with multiple other applications simultaneously, every such discourse is managed strictly in a one-to-one fashion. The state information necessary for every such discourse is held in instances of a general `discourse` class (a CLOS object). The methods defined on this class handle the operations that are applicable to a certain discourse instance. A subclass of `discourse` called `sneps-discourse` handles KQML discourse carried out with SNePS applications. Implementations of KQML performatives that have to perform operations that are SNePS specific are defined as methods on this subclass. This design should make it possible to extend the applicability of the KQML interface to a new knowledge representation system (KRS) by simply adding another subclass to `discourse` whose methods handle the KQML performatives that have to interact with the new KRS in a special way.

3 General Assumptions

Throughout this document all variables, functions, macros, classes, methods and symbols of any kind are assumed to be in the `KQML` package unless explicitly noted otherwise. Only a small portion of these symbols are exported from the `KQML` package (refer to the file `packages.lisp` for more detailed package information). Hence, to use these functions they either have to be within the scope of an `(in-package "KQML")` declaration, or they have to be qualified with the package `KQML` and the double colon syntax.

4 Class Definitions

Instances of the following classes with their associated methods define how discourse between a local and a remote KQML application is carried out:

discourse [*Class*]
This class describes KQML discourse of the most general kind. It has the following instance variables:

request-content-language – Contains a string that specifies the language used in content sentences of requests. The default value is **"interlingua"**.

reply-content-language – Contains a string that specifies the language used in content sentences of replies. If this instance variable is unbound its value defaults to the value of **request-content-language**. The standard method **slot-unbound** has been redefined for instances of class **discourse** to achieve this default behavior.

local-content-language – Contains a string that specifies the language used for content sentences by the local application (or provider).

local-address – A string that specifies the Internet address of the local application.

remote-address – A string that specifies the Internet address of the remote application of this particular discourse.

stream – The Internet stream object via which the local application can communicate with the remote application. "Uttering" is performed by writing onto this stream, "listening" is done by reading from it.

control – A flag that specifies who is in control of this discourse. If this flag is **t** the local application is in control and can "utter" messages such as requests or replies, if it is **nil** it has to "listen" for the remote application to say something, for example, if the requester waits for replies to a particular request.

history – A list of packages exchanged by the local and remote application. It contains all packages sent and received with the most recent package at the beginning of the list.

sneps-discourse [*Class*]
This class is a subclass of **discourse**. It describes SNePS specific KQML discourse carried out with a SNePS application. It has the following instance variables:

request-content-language – See **discourse**. The default value is **"snepslin"**.

local-content-language – See **discourse**. The default value is **"sneps"**.

variable-mapping – Contains a mapping between SNePSLin variables and SNePS variable nodes. The default value is an empty mapping (mapping is a special datatype specified by the SNePSLin translation package).

discourse-context – Contains the name of a SNePS context used for this particular discourse. Adding or removing sentences from the discourse context will be achieved by adding or removing nodes from the SNePS context specified by this slot. Using a different SNePS context for every discourse instance should make it possible to have multiple and completely separated conversations with a single SNePS knowledge base. The default value is a new symbol created by the Common-Lisp function **gentemp**.

5 Interpretation of Packages

Once two applications have established a discourse stream with each other, they communicate by sending and receiving requests and replies encoded as KQML packages. When an application receives a package from the discourse stream it has to interpret it and perform some actions depending on the type of the content expression of the package. In the implementation of this prototype interface the following strategy has been adopted: For every KQML performative defined in [Shapiro and Chalupsky, 1991] there exists a corresponding CLOS method defined on some subclass of `discourse` with almost identical argument structure which handles all the necessary actions required by that particular performative. I.e., an incoming package of any type can be handled by calling the appropriate KQML performative method with the particular discourse instance and the content arguments as parameters. If the execution of a particular performative generates replies then sending of these replies will also be handled by the KQML performative method. The various KQML performative methods and related auxiliary methods are described below.

KQML performatives have to be defined with the following construct:

`defperformative` *name* **`&rest`** *definition* [*Macro*]
The syntax of *definition* is identical to that of a normal CLOS method definition. Using this definitional construct will add *name* to the following variable:

`*performatives*` [*Variable*]
The set of names stored in this variable is used by the package interpreter to determine whether a particular KQML message refers to a bonafide performative.

5.1 General `discourse` methods

As already mentioned, there are two types of methods: KQML performative methods which interpret a particular KQML message (request or reply) whose `:TYPE` slot has the name of the method as its value, and standard methods. Both types are implemented as standard CLOS methods.

`valid-request-language` (*discourse* **`discourse`**) [*Method*]
language
Returns true if *language* is a valid language for content sentences in requests. The only requirement for general discourse is that *language* is a string.

`valid-reply-language` (*discourse* **`discourse`**) [*Method*]
language
Returns true if *language* is a valid language for content sentences in replies. The only requirement for general discourse is that *language* is a string.

`declare-content-languages` (*discourse* **`discourse`**) [*KQML performative*]
`&key` *request-content-language*
reply-content-language
id
Handles a KQML message of that `:TYPE`. It checks whether the specified languages are valid (using the methods described above), and if so assigns them to the according instance variables of *discourse*. *id* is the ID of the package which had this request as its content expression. All KQML methods that handle requests are required to have this argument (see section on error handling).

`send-success-reply` (*discourse* `discourse`) [*Method*]
`&key` *value*
request-id
explanation

Generates a KQML message of `:TYPE success-reply` from the supplied arguments and sends it back to the requester using the information stored in *discourse*.

`success-reply` (*discourse* `discourse`) [*KQML performative*]
`&key` *value*
request-id
explanation
id

Handles a KQML message of that `:TYPE`. If *explanation* was supplied it simply prints that string. Its main purpose is to reset the value of the `control` flag of *discourse* to tell the local application that a certain request has been handled completely.

`send-content-reply` (*discourse* `discourse`) [*Method*]
`&key` *request-id*
reply-number
reply-content-language
content

It translates all sentences in *content* from the local content language into the language specified by *reply-content-language* or the according slot value of *discourse* (using the method `translate-reply-content`), and sends a KQML message of `:TYPE content-reply` generated from the translated content sentences and the other arguments back to the requester.

`content-reply` (*discourse* `discourse`) [*KQML performative*]
`&key` *request-id*
reply-number
reply-content-language
content
id

Handles a KQML message of that `:TYPE`. It uses the `assert` KQML performative method to assert all sentences in *content* in the local discourse context (how KQML performative methods can be used locally for things other than interpreting incoming packages is explained in Section 8).

5.2 SNePS Specific sneps-discourse Methods

The following methods handle SNePS specific KQML discourse. They are defined on the class `sneps--discourse`.

`valid-request-language` (*discourse* `sneps-discourse`) [*Method*]
language

Returns true if *language* is either `"snepslin"` or `"snepsul"`.

`valid-reply-language` (*discourse* `sneps-discourse`) [*Method*]
language

Returns true if *language* is either `"snepslin"` or `"snepsul"`.

translate-request-content (*discourse* **sneps-discourse**) [*Method*]
&key *content*
content-language

Translates *content* (a string specifying a SNePS node) into a SNePSUL (the SNePS User Language) representation of the node (a list). The SNePSUL representation looks just like the SNePSUL command that would have been used to **find** or **assert** the node in question, but with the command name removed. Hence, depending on the performative in question, the SNePSUL command that implements the performative in SNePS can be built by simply consing the appropriate command into the list returned by this method.

If the request content language is **"snepsul"**, then the translation is obtained by simply reading the *content* string using the proper package and read table. If it is **"snepslin"**, then the translation is obtained by using the translation function provided by the SNePSLin package. For SNePSLin sentences, all previously undefined relations that occur in the SNePSUL translation of the sentence will be implicitly defined before the translated sentence is returned. This is necessary, because SNePS requires all relations referenced in a SNePSUL command to be defined.

translate-reply-content (*discourse* **sneps-discourse**) [*Method*]
&key *content*
content-language

Translates *content* (a SNePS node) into a sentence of *content-language*. If the specified content language is **"snepsul"**, then simply a list containing the node and the (flat) cable set that defines it will be returned. For SNePSLin, the translation is obtained by using the back-translation function provided by the SNePSLin package.

set-discourse-context (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
(*content* **'all**)
id

Handles a KQML message of that **:TYPE**. Translates *content* according to *request-content-language* or the request content language specified for *discourse* (using the method **translate-request-content**), and modifies the SNePS context specified for *discourse* with help of the function **modify-sneps-context**.

add-to-discourse-context (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
(*content* **'all**)
id

Handles a KQML message of that **:TYPE**. Similar to **set-discourse-context**.

remove-from-discourse-context (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
(*content* **'empty**)
id

Handles a KQML message of that **:TYPE**. Similar to **set-discourse-context**.

assert (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
content
id

Handles a KQML message of that **:TYPE**. Translates *content* and uses the SNePS function **assert** to assert the translated content sentence in the SNePS context specified for *discourse*. This method assumes that all SNePS relations referenced in the SNePSUL specification of *content* are already defined.

interpret-literally (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
content
id

New KQML message that literally interprets *content* as a SNePSUL command. This method is a low-level mechanism that handles all situations that are not taken care of by standard KQML performatives, for example, it can be used to define new relations if the request content language is SNePSUL.

assign-truth-value (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
truth-value
content
id

Handles a KQML message of that type. In the current implementation it simply replies with a failure message, because SNePS does not deal with truth values. Another way of dealing with this performative in SNePS would be to assert *content* or its negation depending on *truth-value*.

The rest of the KQML performatives deal with different forms of question answering. All of them make use of the following workhorse:

infer-answers (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
reply-content-language
(*worklevel* '**minimal**)
(*how-many* **1**)
(*report-mode* '**continuous**)
(*truth-values* '**any**)
(*inference-mode* '**backward**)
topics
id

Does the actual work for forward, backward and topical queries. It tries to infer answers related to *topics* by first translating them into SNePS nodes and then performing either forward or backward inference on these nodes in the order given in *topics*.

If *inference-mode* is **backward** then the sentences in *topics* are considered queries that should be answered one after the other by asking them using the SNePS function **deduce**. If *worklevel* is **minimal**, then no actual deduction is done, but only nodes that are already in the network and that match the translated content will be returned. If *worklevel* is **maximal**, then an unrestricted deduction is started to generate answers. The SNePS function **deduce** is called such that it returns control every time one new answer is generated. If according to *how-many* enough new answers were generated then a content reply message with the new answers will be sent to the requester. If the value of *report-mode* is **continuous** then inference will continue from where it was suspended to report the answers. The SNePS inference mechanism has been modified to allow proper suspension and continuation of forward and backward inference (see Section 6). If the value of *report-mode* is **suspend** then the deduction will be suspended. The **control** performative to continue a suspended deduction has not yet been implemented, however, the suspend mode can be used to limit the number of answers generated.

If *inference-mode* is **forward** then the sentences in *topics* are considered to be seeds that should trigger forward inference using the SNePS function **add**. Once all answers triggered from adding one of the topic nodes to the network have been generated, the next node will be added and so on until all nodes were added. For forward inference in SNePS a minimal worklevel does not make sense, hence, forward inference is always done unrestrictedly and the value of *worklevel* will be ignored. Because the SNePS function **add** does not deal with limitation of the number of answers generated (as does **deduce**), a new function **add-suspendingly** has been written to deal with suspending forward inference (see Section 6). The values

of *how-many* and *report-mode* control the reporting of answers just as described for backward inference.

In both forward and backward inference the value of *truth-values* is always ignored.

query-sentence-status (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
reply-content-language
(*worklevel* '**minimal**)
(*how-many* **1**)
(*report-mode* '**continuous**)
(*truth-values* '**any**)
content
id

Handles a KQML message of that **:TYPE**. It calls **infer-answers** in backward inference mode with a topic set that contains *content* as its only element.

query-about-topic (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
reply-content-language
(*worklevel* '**minimal**)
(*how-many* **1**)
(*report-mode* '**continuous**)
(*truth-values* '**any**)
content
id

Handles a KQML message of that **:TYPE**. It first translates *content* and tries to generate a set of questions that might be relevant to ask to get topically related answers. It does so by matching the translated topic node to the network. For every matching node that is not a variable it collects all its dominating nodes and applies the substitution generated by the match. All nodes generated in this way will be added to a set of relevant questions, which will then be supplied to **infer-answers** as a set of questions to be answered by backward inference. Currently, atomic topic phrases can not be handled.

assert-and-infer (*discourse* **sneps-discourse**) [*KQML performative*]
&key *request-content-language*
reply-content-language
(*worklevel* '**minimal**)
(*how-many* **1**)
(*report-mode* '**continuous**)
(*truth-values* '**any**)
(*assertion-mode* '**actual**)
content
id

Handles a KQML message of that **:TYPE**. It first asserts all sentences in *content* in the current discourse context using the KQML performative method **assert**, and then calls **infer-answers** in forward inference mode with these sentences as seeds. If *assertion-mode* was **actual**, all added nodes and derived results will be kept in the discourse context. If assertion mode was **hypothetical** the added nodes will be removed from the discourse context after inference and reporting of results has completed, and by that, all derived results will automatically be removed, too.

6 Inference Control

As mentioned above, the SNIP inference mechanism (SNIP stands for SNePS Inference Package) had to be augmented so that inference will get suspended whenever a certain number of new answers has been generated. Inference suspension has to deal with two problems: 1) At what points to suspend the inference, and 2) saving enough state information such that inference can be continued from where it was suspended. The way SNIP is built on top of a multi-processing system called Multi makes the second task rather easy. All that has to be done is to save the process queues when inference gets suspended. To find out the right point for suspension of inference is rather easy for backward inference, and slightly trickier for forward inference.

suspend-inference *multi:*NAME** [*Function*]
Saves all the necessary status of the Multi system such that inference can be properly continued from the point of suspension. This function can also be run as a Multi process. If it gets scheduled at the front of the high-priority queue it can be used to suspend inference immediately after the current process terminates.

6.1 Suspending Backward Inference

Suspending backward inference can naturally be handled by the number-of-answers argument given to **deduce**. Whenever a new answer gets generated it gets reported to the User process (a special Multi process). Once the User process has enough answers accumulated it calls the function **snip::suspend--inference** and then terminates. All we had to do is to redefine this function to call **suspend-inference** to save all the necessary state information. The standard way in which suspending backward inference is started, is to issue a call of the form (**deduce 1 ...node description...**). Further results can then be obtained by repeatedly using the command **continue-inference** described below.

6.2 Suspending Forward Inference

Implementing suspending forward inference is slightly more complicated, because here we do not have the concept of an answer to a query. Whenever a rule fires and possibly asserts a new node, we have a possible new answer that should get reported and inference suspended. Hence, a natural point to check for newly derived answers is in the function that broadcasts a single report. The redefined function checks whether suspending forward inference is in progress, and if so whether after broadcasting the report a new node has been added to the network. If that was the case, it schedules a **suspend-inference** process that will save all the necessary state and suspend inference right after the current process terminates.

add-suspendingly **&rest** *snd* [*SNePSUL Command*]
Adds the node defined by the SNePSUL node description *snd* to the network and starts suspending forward inference on it. This function will immediately return by reporting the added node. Further results can be obtained by repeatedly executing the **continue-inference** command.

continue-inference () [*SNePSUL Command*]
Determines whether the immediately preceding inference was forward or backward inference, and then continues the suspended inference accordingly until it suspends inference again at the next logical suspension point. The set of new answers will be returned. Once all inference possibilities have been exhausted and no more answers can be generated, NIL will be returned.

7 KQML Networking and Client/Server Protocol

KQML applications can communicate with each other over the Internet using the TCP/IP protocol. Every application is identified by an optional application identifier and by an Internet address, for example, **app@hadar.cs.buffalo.edu**, or, if there is only one KQML application running on a particular machine, just **hadar.cs.buffalo.edu** would be sufficient.

7.1 Enabling and Disabling the KQML Service

An application that wishes to act as a KQML server can enable and disable the KQML service with help of the following functions:

enable-kqml-service **&optional** *application-id* (*discourse-type* **'discourse**) [*Function*]
Defines a KQML service with name **:KQML-***application-id* or just **:KQML** if no application id was supplied. For example, if the value of *application-id* was **"App"**, then the service name will be **:|KQML-App|**. Associated with this name will be a KQML server function which will be started in a separate process once a client has connected to this service successfully. *discourse-type* determines the type of the discourse instance created by the server to manage the state of the discourse. **enable-kqml-service** also starts a demon process that monitors the Internet for KQML service requests.

The actual guts of the TCP/IP networking are handled by a **tcp.lisp** package written by Rich Fritzson. The way services are defined is derived from that package.

disable-kqml-service **&optional** *application-id* [*Function*]
Disables the KQML service identified by *application-id*.

7.2 Connecting to a KQML Server

A client can establish a connection to a KQML server with the following functions:

connect-to-kqml-server *address* [*Function*]
Opens and returns a bidirectional TCP/IP stream to the KQML server running at *address*. It first tries to establish a TCP/IP connection to the remote process. Then it derives a KQML service name from the address and submits that as the first symbol to the remote process. If a KQML service of that name has been enabled at the remote host the KQML server there will start up and acknowledge the service request. Once the acknowledgment has been received the stream object will be returned. If no connection could be made (e.g., if the address did not exist, or no service of that name was enabled at the remote host) then an error message will be printed and NIL will be returned. Currently, the waiting for an acknowledgment from the KQML server does not time out. It should.

open-kqml-discourse *address* **&optional** (*discourse-type* **'discourse**) [*Function*]
This is the high-level interface to establish a discourse with a KQML server at *address*. It uses **connect-to-kqml-server** to connect to the remote process, and, if successful, creates and initializes a discourse instance of type *discourse-type* which will be returned.

close-kqml-discourse *discourse* [*Function*]
Terminates a KQML discourse by closing the TCP/IP stream stored in *discourse*.

7.3 Sending and Receiving Packages

make-package-id *address* [*Function*]
Generates a unique package identifier (a string) from *address* and a local time stamp.

send-kqml-package *package discourse* [*Function*]
Delivers *package* to the stream specified in *discourse*, and stores it at the front of the history stored in the **history** slot of *discourse*.

send-request *content discourse* [*Function*]
Creates a package with *content* as its content expression, proper **:ID**, **:FROM**, **:TO**, etc., slots, and delivers it to the stream specified in *discourse*.

send-reply *content discourse* [*Function*]
Extracts the value of the **:REQUEST-ID** slot from *content* and tries to find a request package in the history of *discourse* which has that id. If a proper request could be found, it creates a package with *content* as its content expression, proper **:ID**, **:FROM**, **:TO**, etc., slots, and delivers it to the stream specified in *discourse*. The history check makes sure that no unsolicited replies will be sent.

7.4 Input Interpretation

Tokens transmitted across a TCP/IP stream are divided into the following classes:

Symbols are used to communicate simple commands or states, or to identify services.

Strings are used to transmit low-level error messages.

Packages , i.e., lists for which the predicate **is-package** is **t**, are used to transmit KQML messages.

These tokens are read from a TCP/IP stream with the Common-Lisp **read** function, classified by their type and interpreted accordingly. All input that does not fall in any of the above categories is considered illegal.

interpret-tcp-stream-input *input discourse* [*Function*]
Does the low-level interpretation of *input* which is a single token read from the TCP/IP stream specified in *discourse*. It classifies the token according to the categories described above and interprets it accordingly. If the token signals a closing operation or eof, the local stream object will be closed, too. If it was an error message the message will be printed in the local process. If it was a package it will be submitted to the function **interpret-package**. Otherwise, an illegal input error will be signalled to the remote process.

interpret-package *package discourse* [*Function*]
Performs the high-level interpretation of a package token. It checks whether *package* contains a defined performative in its content slot, and if so constructs a proper argument list from the content of the package and *discourse*, to which then the appropriate KQML performative will be applied.

7.5 Client/Server Discourse

Once a connection between a client and a server has been established a complete discourse transaction is defined by the following protocol (for synchronous and blocking communication):

Client Application	*KQML server*
	Wait (read) for next request
Send (write) a request Wait for next reply	
	Interpret request Send 0 or more content replies without waiting inbetween
Read and interpret 0 or more content replies	
	Send a success reply
Read and interpret success reply and regain discourse control	
	Wait for next request
	.....

After a connection between a client and a server has been established, the KQML server stays in a read- request/interpret-request/send- replies loop until the connection gets closed by the remote process, or some unrecoverable error occurs. The waiting and blocking is done by the Common-Lisp **read** function. Because a certain request can trigger multiple content reply messages, there must be a way to tell the client when it cannot expect any more replies. For this purpose the following policy has been adopted:

> Every KQML performative method has to terminate by sending exactly one **success-reply** package back to the client, regardless of whether the request triggered any content replies or not.

Using this policy, after issuing a request the client can simply read and interpret successive incoming packages until it finally receives a success reply package which will result in the regaining of discourse control. As a corollary to this policy, in the case of an error during the handling of a request the KQML server must send an error message back to the client to inform it that something went wrong, otherwise the client would wait indefinitely.

On the other hand, the KQML server does not wait for any acknowledgment from the server. Because multiple content replies for a certain request might be generated at a faster rate than the client can process them, it is assumed that the underlying TCP/IP implementation does the necessary buffering and blocking.

kqml-server *stream remote-host application-id discourse-type* [*Function*]
This function implements the server side of the protocol. Once a *remote-host* has established a TCP/IP connection, the demon function starts the KQML server function in a separate process (see **enable-kqml--service** and **open-kqml-discourse**). The server then creates and initializes a discourse object of type *discourse-type*, acknowledges a successful connection back to the remote host, and then starts reading from *stream* thus waiting for the first request. It terminates if the stream gets closed by the remote host, or if some unrecoverable error occurs.

send-request-and-interpret-replies *content discourse* [*Function*]
This function implements the client side of the protocol. It generates a package with *content* as its content expression and delivers it to the KQML server specified by the stream stored in the *discourse* object. It then sets the **control** slot of *discourse* to **nil** and reads and interprets reply packages until the value of that slot is **t** again (to which it will be set once the **success-reply** method has been executed as a result of receiving a package of this type). Once discourse control has been regained, or an error has been signalled by the KQML server, this function returns.

8 Local Execution

Sometimes it is desirable to use KQML performative methods for local execution only, and not just to interpret a certain package received from a remote application. For example, the **content-reply** and **assert-and-infer** methods use the **assert** method. To avoid infinite request/reply cycles (the policy stated above says that every performative method has to terminate by sending a **success-reply** message back) the following macros are available:

unless-only-locally **&body** *body* [*Macro*]
Forms within the scope of this macro will not get executed in local execution mode. The **send-success--reply** and **send-content-reply** methods use this macro to avoid sending replies from locally executed performatives.

execute-locally **&body** *body* [*Macro*]
Within the scope of this macro forms that are wrapped within the **unless-only-locally** macro will not get executed.

9 The SNePSLin Package

SNePSLin is a linear representation for SNePS nodes. Because SNePS nodes are defined as cable sets, the syntax of SNePSLin is very similar to the syntax of cable sets. A SNePSLin sentence can be used to linearly specify a SNePS node or pattern that should be asserted, queried, etc. Here is a syntax specification for SNePSLin:

```
<snepslin sentence> ::=  ( <snepslin cable> .... )
<snepslin cable> ::= (<downrel> <snepslin node> .... <snepslin node>)
<downrel> :: A SNePS downward relation
<snepslin node> ::= <snepslin sentence> | <base node> | <var node>
<base node> ::= <implicit base node> | <new base node>
<implicit base node> ::= <symbol>
<new base node> ::= +<symbol>
<var node> ::= <implicit var node> | <new var node>
<implicit var node> ::= ?<symbol>
<new var node> ::= +?<symbol>
<symbol> ::= any Common-Lisp symbol
```

"Implicit" variable or base nodes refer to a previously existing node with the same name, otherwise they implicitly define a new variable or base node.

"New" nodes will create a new variable or base node that can be referred to by name. The new node mechanism makes sure that the referenced nodes are new and not accidentally identical with any previously existing nodes of that type.

9.1 Naming Maps

Naming maps provide proper translation between SNePSLin atoms and SNePS atoms (i.e., variable and base nodes), for example, **+?person** will create a new SNePS variable node - say **v10** - that can be referred to as **?person** in subsequent SNePSLin expressions. **+arbindv** will create a new SNePS base node - say **b52** - that can be referred to as **arbindv**.

A special data structure called *mapping* is used to store these one-to-one naming maps. Mappings are implemented as structures that contain two hash tables, one for mapping SNePSLin atoms into SNePS atoms, and one for mapping SNePS atoms back into SNePSLin. So far, only variables are mapped. The variable mapping for a particular discourse is stored in the according slot of the discourse instance.

9.2 Translation Functions

The following functions define the top-level translation interface between SNePSLin and SNePSUL:

snepslin-to-snepsul *snepslin-sentence discourse* [*Function*]
Translates *snepslin-sentence* (a list) into a SNePSUL description (a list) of the node specified by the SNePSLin sentence. The SNePSUL description can then be used to assert/deduce etc. that node by simply putting the actual SNePSUL command name as the first element of the list and executing the resulting command. *discourse* is used to specify the proper naming maps for the translation of SNePSLin atoms.

sneps-to-snepslin *sneps-node discourse* [*Function*]
Takes an actual *sneps-node* and translates it into a SNePSLin specification (a list) according to the naming maps of *discourse*. This function replaces every molecular node in the downward fcableset of *sneps-node* by its SNePSLin representation, hence it cannot deal with circular networks (in SNePS-2.1 there are no cycles in the network, hence, this is not a problem).

10 Error Handling, Tracing and Debugging

A robust KQML provider has to be able to handle error conditions gracefully. The fact that the actual knowledge base that provides information is a complex and slightly fragile Lisp program makes error handling particularly important. The Common-Lisp condition system provides an elegant mechanism to deal with exceptions. The current implementation distinguishes between four kinds of errors:

KQML errors: These are errors caused by KQML messages of erroneous content, for example, if a content language was requested which cannot be handled by the local provider.

KQML system errors: These are errors that are due to bugs in the KQML interface software.

SNePS errors: These are errors caused by erroneous SNePSUL commands, for example, if a command references a previously undefined relation.

SNePS system errors: These are errors due to bugs in the SNePS software.

with-kqml-errors {*form*}* [*Macro*]
Evaluates the *forms* of the body (an implicit **progn**) with proper error handlers installed to handle the types of errors described above. If an error occurs, the appropriate handler for that error is called. These handlers generate a **success-reply** message with **:value** set to **failure** and an appropriate explanation. They assume that the variable **id** is bound to the id of the request in which the error occurred (this macro is only used in KQML message methods that define requests, all of which have an **id** parameter that specifies the id of the request). Because KQML message methods might themselves invoke other KQML message methods to achieve their goal, a special function **invoke-toplevel-restart** has been written to return to the error handlers installed in the outermost (or top-level) invocation of **with-kqml-errors**.

debug [*Variable*]
A non-NIL value of this variable will result in the production of various tracing and debugging messages during the execution of the discourse.

when-debug **&body** *body* [*Macro*]
Forms within the scope of this macro will be executed only if ***debug*** is non-NIL (used for conditionally printing out debugging and tracing information).

11 A SNePSUL Interface to KQML

The following describes a small and very experimental SNePSUL (the SNePS User Language) top level that can be used to establish connections to remote SNePS/KQML servers, and that defines remote versions of standard SNePSUL commands that use KQML messages to communicate with the remote servers.

default-discourse [*SNePSUL Variable*]
Holds the name of the current default discourse (as a one-element list). This name is itself a SNePSUL variable whose value is the actual discourse object. The default value of **default-discourse** is **(default-default-discourse)**.

open-sneps-discourse *address* **&optional** *discourse-name* [*SNePSUL Command*]
Creates a connection to a KQML server at *address*, assigns the corresponding discourse object to the SNePSUL variable *discourse-name*, and makes this discourse name the default discourse by assigning it to the SNePSUL variable **default-discourse**. If no discourse name has been supplied it defaults to ***default-discourse**. The discourse will be initialized by declaring SNePSLin as the content language, and by setting the remote discourse context to the empty context. Returns the value of *discourse-name*.

close-sneps-discourse *address* **&optional** *discourse-name* [*SNePSUL Command*]
Terminates the discourse identified by *discourse-name* (defaults to ***default-discourse**).

Remote SNePSUL commands generally have the same argument structure as their standard SNePSUL counterparts, however, they have an additional optional **:discourse** argument that allows one to supply the name of a discourse (similar to a context specification). An also optional **:context** argument can be used to specify in what context answers should be asserted. Both of these arguments have to be at the end (or far right) of the arguments supplied to the remote command. If omitted the context defaults to ***defaultct**, and the discourse to ***default-discourse**. Here is an example call:

```
(remote-deduce member $who class mortal :discourse mydiscourse)
```

remote-assert **&rest** *node-spec* **&optional** *discourse-spec* [*SNePSUL Command*]
Asserts a node defined by *node-spec* in the remote discourse context of the discourse specified by *discourse-spec*. (Note, the combination of **&rest** and **&optional** keywords in the above specification is not legal Common-Lisp, their intended meaning, however, should be clear from the preceding paragraph). Specifying a context for this command is a noop.

remote-findassert **&rest** *node-spec* **&optional** *discourse-context-spec* [*SNePSUL Command*]
Tries to find asserted nodes that match the node defined by *node-spec* in the remote discourse context of the discourse specified by *discourse-context-spec*. Answers will be asserted as new hypotheses in the local discourse context. This is different from the semantics of the standard **findassert** command which just searches the network, but does not build anything! Returns the set of newly asserted nodes that were not already part of the network.

remote-deduce **&rest** *node-spec* **&optional** *discourse-context-spec* [*SNePSUL Command*]
Tries to find answers to the query defined by *node-spec* in the remote discourse context of the discourse specified by *discourse-context-spec* by way of backward inference. Answers will be asserted as new hypotheses in the local discourse context, rather than as derived nodes as done by the standard **deduce**. Returns the set of newly asserted nodes that were not already part of the network.

remote-add **&rest** *node-spec* **&optional** *discourse-context-spec* [*SNePSUL Command*]
Tries to infer new nodes from the node defined by *node-spec* in the remote discourse context of the discourse specified by *discourse-context-spec* by way of forward inference. Answers will be asserted as new hypotheses in the local discourse context. Returns the set of newly asserted nodes that were not already part of the network.

New remote SNePSUL commands can be defined with the following macro:

def-snepsul-remote *command* **&rest** *definition* [*Macro*]
Defines a SNePSUL command with name **remote-***command* according to *definition*. The new remote command inherits all command properties from its standard SNePSUL *command* brother. All necessary SNePSUL package import/export will be done automatically.

11.1 An Example Demonstration Using Remote Commands

The following is a short example demo that shows how the remote commands described above can be used at the SNePS top level. The demo was run in a Lisp process on **pegasus.cs.buffalo.edu**, while the KQML server was run on **hadar.cs.buffalo.edu**.

```
> (sneps)

   Welcome to SNePS-2.1

 Copyright 1984, 88, 89 by Research Foundation of State University of New York

   5/15/1992 16:31:10

* (demo "~/Kqml/Code/remote-demo.sneps21")


File /u6/grads/hans/Kqml/Code/remote-demo.sneps21 is now the source of input.


 CPU time : 0.09     GC time : 0.00

* ;; This demo assumes that a KQML server is running in a Lisp
;; process on hadar.cs and a service has been enabled with
;; (enable-kqml-service "Test")
;;
(resetnet t)

Net reset


 CPU time : 0.02     GC time : 0.00

* (^ (setq kqml::|*DEBUG*| nil))
```

```
 CPU time : 0.00     GC time : 0.00

* ;; Need to define these relations because the nodes in the
;; remote commands have to be built before they can be
;; translated into SNePSLin. New relations in answers will
;; be defined implicitly by the assert performative.
(define member class)

(MEMBER CLASS)

 CPU time : 0.03     GC time : 0.00

* (open-sneps-discourse "Test@hadar.cs")

(DEFAULT-DEFAULT-DISCOURSE)

 CPU time : 0.18     GC time : 0.00

* ;; Remote assertions do not return anything, because they
;; build nodes in the remote discourse context
(remote-assert member hans class man)


 CPU time : 0.09     GC time : 0.00

* (remote-assert member otto class man)


 CPU time : 0.08     GC time : 0.00

* (remote-assert forall $x ant (build member *x class man)
       cq  (build member *x class human))


 CPU time : 0.12     GC time : 0.00

* (remote-assert forall $x ant (build member *x class human)
       cq  (build member *x class mortal))


 CPU time : 0.12     GC time : 0.00

* (full-describe (remote-deduce member $who class mortal))

(M5! (CLASS MORTAL)
     (MEMBER OTTO)
     (HYP ((ASSERTIONS (M5!)) (RESTRICTION NIL)
                              (NAMED NIL))))
(M6! (CLASS MORTAL)
     (MEMBER HANS)
     (HYP ((ASSERTIONS (M6!)) (RESTRICTION NIL)
                              (NAMED NIL))))

(M5! M6!)
```

```
 CPU time : 0.33     GC time : 0.00

* (describe *nodes)

(HANS)
(M5! (CLASS MORTAL) (MEMBER OTTO))
(M6! (CLASS MORTAL) (MEMBER HANS))

(HANS M5! M6! MORTAL OTTO)

 CPU time : 0.05     GC time : 0.00

* ;; This will not return the originally added node, because that
;; was already in the network (unasserted, though)
(full-describe (remote-add member deepak class man))

(M8! (CLASS HUMAN)
     (MEMBER DEEPAK)
     (HYP ((ASSERTIONS (M8!)) (RESTRICTION NIL)
                              (NAMED NIL))))
(M9! (CLASS MORTAL)
     (MEMBER DEEPAK)
     (HYP ((ASSERTIONS (M9!)) (RESTRICTION NIL)
                              (NAMED NIL))))

(M8! M9!)

 CPU time : 0.40     GC time : 0.00

* ;; After the remote-add it is asserted, too
(describe *nodes)

(DEEPAK)
(HANS)
(HUMAN)
(M5! (CLASS MORTAL) (MEMBER OTTO))
(M6! (CLASS MORTAL) (MEMBER HANS))
(M7! (CLASS MAN) (MEMBER DEEPAK))
(M8! (CLASS HUMAN) (MEMBER DEEPAK))
(M9! (CLASS MORTAL)
     (MEMBER DEEPAK))

(DEEPAK HANS HUMAN M5! M6! M7! M8! M9! MAN MORTAL OTTO)

 CPU time : 0.09     GC time : 0.00

* (close-sneps-discourse)

(DEFAULT-DEFAULT-DISCOURSE)

 CPU time : 0.02     GC time : 0.00

*

End of /u6/grads/hans/Kqml/Code/remote-demo.sneps21 demonstration.
```

References

[Shapiro and Chalupsky, 1991] Stuart C. Shapiro and Hans Chalupsky. KQML - issues and review. Preliminary report, Department of Computer Science, State University of New York at Buffalo, Buffalo, NY, Dec 1991.

[Shapiro and Rapaport, 1987] S. C. Shapiro and W. J. Rapaport. SNePS considered as a fully intensional propositional semantic network. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 263–315. Springer–Verlag, New York, 1987.

[Shapiro and Rapaport, 1992] Stuart C. Shapiro and William J. Rapaport. The SNePS family. *Computers & Mathematics with Applications*, 23(2–5):243–275, January–March 1992.

[Shapiro, 1979] S. C. Shapiro. The SNePS semantic network processing system. In N. V. Findler, editor, *Associative Networks: The Representation and Use of Knowledge by Computers*, pages 179–203. Academic Press, New York, 1979.

[Steele, 1990] Guy L. Steele. *COMMON LISP*. Digital Press, second edition, 1990.

A Appendix

A.1 Installation

To install the KQML prototype interface you need a machine that runs Sun Common Lisp 4.0.1 or higher under the Unix[1] operating system. The installation requires the files listed below, as well as a recent version of the SNePS-2.1 system:

```
tcp.c
load-kqml.lisp
packages.lisp
foreign-support.lisp
tcp.lisp
error-handling.lisp
snepslin.lisp
package-access.lisp
discourse.lisp
kqml-server.lisp
inference-control.lisp
kqml-performatives.lisp
remote-sneps.lisp
```

To install the interface you first have to copy all these files into a single directory, for this example let us call it `/kqml/interface`. Then compile `tcp.c` by doing

```
cc -c tcp.c
```

After that edit the variable `*kqml-directory*` in the file `load-kqml.lisp` to correspond to your installation directory, for our example give it the value `"/kqml/interface"`. Then startup Lisp, load the SNePS system and then compile and load the interface by loading the file `load-kqml.lisp`, for example, issue the following command:

```
(load "/kqml/interface/load-kqml.lisp")
```

The first time around this will compile all the Lisp files before they are loaded, during future loads only the compiled files will get loaded.

[1] Unix is a trademark of AT&T Bell Laboratories

A.2 File load-kqml.lisp

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(in-package :user)


(unless (find-package "SNEPS")
  (setq *sneps21-verbose* t)
  (load "~snwiz/bin/sneps21"))

(defvar *kqml-directory* "~hans/Kqml/Code")

(defvar *kqml-translations*
  `(("**;*.*.*" ,(format nil "~a/**/" *kqml-directory*))
    ("*.*.*" ,(format nil "~a/" *kqml-directory*))
    ))

(setf (lp:logical-pathname-translations "kqml")
      *kqml-translations*)

(make-simple-system
 "KQML"
 '(;; Need this until with-snepsul.lisp is included in the SNePS release.
   (:compile-load "/u6/rstaff/snerg/src/Sneps21/All-Lisps/Work/with-snepsul")
   (:load "kqml:packages")
   (:compile-load "kqml:foreign-support")
   (:compile-load "kqml:tcp")
   (:compile-load "kqml:error-handling")
   (:compile-load "kqml:snepslin")
   (:compile-load "kqml:package-access")
   (:compile-load "kqml:discourse")
   (:compile-load "kqml:kqml-server")
   (:compile-load "kqml:inference-control")
   (:compile-load "kqml:kqml-performatives")
   (:compile-load "kqml:remote-sneps")
   )
 :mode :COMPILE
 :verbose *load-verbose*)
```

A.3 File packages.lisp

```
;;;
;;; RCS: $Header: /u6/grads/hans/Kqml/Code/packages.lisp,v 1.1 92/05/15 15:27:54 hans Exp Locker: hans $
;;;


(in-package "USER")

;; Various package definitions. All these definitions "play it safe" by
;; using only strings in the definitions, hence no spurious symbols in
;; the user package will be created during the load of this file.

;; from ff.lisp:
(defpackage "FF"
  (:use "LISP" "LUCID-COMMON-LISP") ;; No package COMMON-LISP, sigh
  (:export #:malloc-foreign-string
           #:errno
           #:reset-errno
           #:sys_errlist
           ))

;; from tcp.lisp:
(defpackage "TCP"
  (:use "LISP" "LUCID-COMMON-LISP" "FF")
  (:export #:completed-p #:result #:body #:from
           #:deliver-to
           #:receive-from
           #:connect-to-service
           #:start-server
           ))

(defpackage "KQML"
  (:use "LISP" "LUCID-COMMON-LISP")
  (:shadow #:assert)
  (:export #:implicit-var #:implicit-base #:continue-inference
           #:add-suspendingly #:open-sneps-discourse
           #:close-sneps-discourse
           )
  )

;; Import KQML things needed by SNePSUL:
(import '(kqml:implicit-var kqml:implicit-base kqml:continue-inference
          kqml:add-suspendingly kqml:open-sneps-discourse
          kqml:close-sneps-discourse
          ) 'snepsul)
```

A.4 File error-handling.lisp

```
;;;
;;; RCS: $Header: /u6/grads/hans/Kqml/Code/error-handling.lisp,v 1.1 92/05/15 15:29:29 hans Exp Locker: hans $
;;;

(in-package "KQML")

;; Error Handling stuff:

(defvar *debug* t
  "If t go into the debugger when system errors occur")

(defmacro when-debug (&body body)
  `(when (and (boundp '*debug*) *debug*)
     ,@body))

;; Need this because in case of an error I always want to return to
;; the outermost KQML message method in which the error occured.
(defun invoke-toplevel-restart (restart-name &rest args)
  "Invokes the outermost (or toplevel) restart with RESTART-NAME"
  (let ((restart (find-if #'(lambda (restart)
                              (equal (restart-name restart) restart-name))
                          (reverse (compute-restarts)))))
    (apply #'invoke-restart (cons restart args))))

(defmacro with-kqml-errors (&body body)
  "Macro to be used in KQML message methods to handle KQML-SYSTEM-ERRORS that
are due to bugs in this software, KQML-ERRORS that handle KQML messages of
erroneous content, SNEPS-ERRORS due to erroneous commands supplied to SNePS,
and SNEPS-SYTEM-ERRORS due to bugs in the SNePS system. In case of any
such error a success-reply message will be sent that describes the error.
Assumes variables DISCOURSE and ID to be bound properly."
  `(sneps:in.environment
    :functions ((sneps:sneps-error #'sneps-error))
    :eval
    (handler-bind ((error #'(lambda (condition)
                              (invoke-toplevel-restart
                               'kqml-system-error
                               condition))))
     (restart-case (progn ,@body)
      (kqml-system-error (condition)
                         (when-debug (invoke-debugger condition))
                         (send-success-reply
                          discourse :value 'failure :request-id id
                          :explanation
                          (format nil "KQML system error: ~a" condition)))
      (kqml-error (explanation)
                  (send-success-reply
                   discourse :value 'failure :request-id id
                   :explanation
                   (format nil "KQML error: ~a" explanation)))
      (sneps-system-error (condition)
                          (when-debug (invoke-debugger condition))
                          (send-success-reply
                           discourse :value 'failure :request-id id
```

```
                        :explanation
                        (format nil "SNePS system error: ~a" condition)))
      (sneps-error (explanation)
                   (send-success-reply
                    discourse :value 'failure :request-id id
                    :explanation
                    (format nil "SNePS error: ~a" explanation))))))

(defun kqml-error (explanation)
  "Signal a KQML error described by EXPLANATION (a string)"
  (invoke-toplevel-restart 'kqml-error explanation))

;; Redefine this SNePS function for proper SNePS error handling:
(defun sneps-error (msg module fn)
  "Signal a SNePS error"
  (invoke-toplevel-restart
   'sneps-error
   (format nil "~A~%     Occurred inside ~A -- in function ~A~%"
           msg module fn)))
```

A.5 File snepslin.lisp

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; Package: SNEPS; Base: 10 -*-

(in-package "KQML")

(rcs-info "$Header: /u6/grads/hans/Kqml/Code/snepslin.lisp,v 1.3 92/05/15 15:15:58 hans Exp Locker: hans $")


;; SNePSLin is a linear representation for SNePS nodes.
;; Because SNePS nodes are defined as cable sets, SNePSLin is very similar to
;; cable sets. A SNePSLin sentence can be used to linearely specify a SNePS
;; node/pattern that should be asserted, queried etc.

;; Data Type:
;;   <snepslin sentence> ::=  ( <snepslin cable> .... )
;;   <snepslin cable> ::= (<downrel> <snepslin node> .... <snepslin node>)
;;   <downrel> :: A SNePS downward relation
;;   <snepslin node> ::= <snepslin sentence> | <base node> | <var node>
;;   <base node> ::= <implicit base node> | <new base node>
;;   <implicit base node> ::= <symbol>
;;   <new base node> ::= +<symbol>
;;   <var node> ::= <implicit var node> | <new var node>
;;   <implicit var node> ::= ?<symbol>
;;   <new var node> ::= +?<symbol>
;;   <symbol> ::= any Common-Lisp symbol
;;
;; "Implicit" nodes refer to a previously existing node with the same
;; name, otherwise they implicitly create a new node with that name
;; (i.e., nodes are implicitly defined by mentioning them (just like
;; FORTRAN....))
;;
;; "New" nodes will create a new node of that type that can be
;; referred to by name. The new node mechanism makes sure that the
;; referenced nodes are new and not accidentally identical with any
;; preexisting nodes of that type.

;; A naming map will provide proper translation between SNePSLin atoms
;; and SNePS atoms (i.e., variable and base nodes), e.g., +?person
;; will create a new variable node - say v10 - that can be referred to
;; as ?person. +arbindv will create a new base node - say b32 - that
;; can be referred to as arbindv.


(defun snepslin-to-snepsul (snepslin-sentence discourse)
  "Translates a SNEPSLIN-SENTENCE (a list) into a SNePSUL description (a
list) of the node specified by the SNePSLin sentence. The SNePSUL
description can then be used to assert/deduce etc. that node by simply
putting the actual SNePSUL command name as the first element of the
list and executing the resulting SNePSUL command. DISCOURSE is used to
find the proper naming maps for the translation of SNePSLin atoms."
  (let (relations)
    (values
     (mapcan
      #'(lambda (snepslin-cable)
          (let ((relation (translate-snepslin-relation-to-sneps
```

```
                        (first snepslin-cable)
                        discourse))
                (nodeset (rest snepslin-cable)))
            (pushnew relation relations)
            (list relation
                  (mapcar
                   #'(lambda (snepslin-node)
                       (cond ((consp snepslin-node)
                              (multiple-value-bind (snepsul-expression
                                                    new-relations)
                                  (snepslin-to-snepsul
                                   snepslin-node discourse)
                                (setq relations (union relations
                                                       new-relations))
                                (cons 'sneps:build snepsul-expression)))
                             (t (translate-snepslin-atom-to-sneps
                                 snepslin-node discourse))))
                   nodeset))))
           snepslin-sentence)
     relations)))

(defun translate-snepslin-atom-to-sneps (snepslin-atom discourse)
  "Translates a SNEPSLIN-ATOM into a SNePS atom depending on the naming
maps of DISCOURSE. This will create new isolated SNePS nodes!!"
  (declare (ignore discourse))
  (let ((name (symbol-name snepslin-atom)))
    (case (elt name 0)
      ;; "New" nodes not handled yet
      ;;(#\+ (case (elt name 1)
      ;;       (#\? (new-var (subseq name 2) discourse))
      ;;       (t (new-base (subseq name 1) discourse))))
      (#\? (implicit-var (subseq name 1) discourse))
      (t (implicit-base name discourse)))))

(defun translate-snepslin-relation-to-sneps (snepslin-relation discourse)
  "Translates a SNEPSLIN-RELATION into a SNePS relation (dummy so far)"
  (declare (ignore discourse))
  snepslin-relation)

;; A simple MAPPING data type that deals with one-to-one mappings:
(defstruct mapping
  (forward (make-hash-table :test #'equal))
  (inverse (make-hash-table :test #'equal)))

(defun initialize-mapping (mapping)
  (clrhash (mapping-forward mapping))
  (clrhash (mapping-inverse mapping)))

(defun lookup (key mapping)
  (gethash key (mapping-forward mapping) :undefined))

(defun lookup-inverse (value mapping)
  (gethash value (mapping-inverse mapping) :undefined))

(defmacro undefined-p (key mapping)
  `(eq (gethash ,key (mapping-forward ,mapping) :undefined)
```

```
      :undefined))

(defmacro undefined-inverse-p (key mapping)
  `(eq (gethash ,key (mapping-inverse ,mapping) :undefined)
       :undefined))

(defun define-mapping (key value mapping)
  (setf (gethash key (mapping-forward mapping)) value)
  (setf (gethash value (mapping-inverse mapping)) key))

;; Variable mappings: string -> varnode
(defun implicit-var (name discourse)
  "Returns the actual SNePS variable node referred to by NAME (a string)
according to the naming map of DISCOURSE. If the value of NAME is undefined
a new variable node will be created to be the value of NAME from then on."
  (let ((variable-mapping (slot-value discourse 'variable-mapping)))
    (cond ((or (undefined-p name variable-mapping)
               ;; A resetnet or erase might have eliminated a varnode
               ;; still stored in a slot of the mapping:
               (not (sneps:ismemb.ns
                     (lookup name variable-mapping)
                     (sneps:value.sv 'sneps:varnodes))))
           (let ((newvar (sneps:choose.ns (sneps:$ 'newvar))))
             (define-mapping name newvar variable-mapping)
             newvar))
          (t (lookup name variable-mapping)))))

(defun implicit-base (name discourse)
  "Returns the actual SNePS base node referred to by NAME (a dummy)."
  (declare (ignore discourse))
  name)

;; Translate a SNePS node into its SNePSLin specification:

(defun sneps-to-snepslin (sneps-node discourse)
  "Takes an actual SNEPS-NODE and translates it into a SNePSLin
specification (a list) according to the naming maps of DISCOURSE. This
function replaces every molecular node in the downward fcableset of
SNEPS-NODE by its SNePSLin representation, hence it cannot deal with
circular networks!!"
  (let ((cable-set (sneps::down.cs
                    (sneps::fcs-to-cs
                     (sneps:node-fcableset sneps-node))))
        snepslin-sentence)
    (cond ((sneps::isnew.cs cable-set)
           (translate-sneps-atom-to-snepslin sneps-node discourse))
          (t (sneps::do.cs (cable cable-set)
               (push (cons (translate-sneps-relation-to-snepslin
                            (sneps:relation.c cable)
                            discourse)
                           (let (snepslin-node-set)
                             (sneps:do.ns (node (sneps:nodeset.c cable))
                                (push (sneps-to-snepslin node discourse)
                                      snepslin-node-set))
                             (reverse snepslin-node-set)))
                     snepslin-sentence))
             snepslin-sentence))))
```

```
            (reverse snepslin-sentence)))))

(defun translate-sneps-atom-to-snepslin (atomic-sneps-node discourse)
  "Translates an ATOMIC-SNEPS-NODE into a SNePSLin atom according to
the naming maps of DISCOURSE."
  (let ((variable-mapping (slot-value discourse 'variable-mapping)))
    (when (and (sneps:isvar.n atomic-sneps-node)
               (undefined-inverse-p atomic-sneps-node variable-mapping))
      (define-mapping
          (symbol-name (gensym "V")) atomic-sneps-node variable-mapping))
    (intern (cond ((sneps:isvar.n atomic-sneps-node)
                   (format nil "?~a"
                           (lookup-inverse
                            atomic-sneps-node
                            (slot-value discourse 'variable-mapping))))
                  (t (symbol-name (sneps:node-na atomic-sneps-node))))
            'kqml)))

(defun translate-sneps-relation-to-snepslin (relation discourse)
  "Translates a SNePS RELATION into a SNePSLin relation."
  (declare (ignore discourse))
  (intern (symbol-name relation) 'kqml))
```

A.6 File package-access.lisp

```
;;; Package access functions:
;;;
;;; RCS: $Header: /u6/grads/hans/Kqml/Code/package-access.lisp,v 1.1 92/05/15 15:30:00 hans Exp Locker: hans $
;;;

(in-package "KQML")


(defmacro new-plist-with-header (header)
  `(list ,header))

(defmacro is-plist-with-header (thing header)
  `(and
    (consp ,thing)
    (eq (car ,thing) ,header)
    (oddp (length ,thing))))

(defmacro new-package ()
  `(new-plist-with-header 'package))

(defmacro is-package (thing)
  `(is-plist-with-header ,thing 'package))

(defmacro get-package-slot (package &rest slots)
  (case (length slots)
    (0 nil)
    (1 `(getf (cdr ,package) ,(first slots)))
    (t `(get-package-slot
         (getf (cdr ,package) ,(first slots))
         ,@(rest slots)))))

(defmacro set-package-slot (package value &rest slots)
  (case (length slots)
    (0 nil)
    (1 `(setf (getf (cdr ,package) ,(first slots)) ,value))
    (t `(set-package-slot
         (getf (cdr ,package) ,(first slots))
         ,value
         ,@(rest slots)))))

(defmacro new-message ()
  `(new-plist-with-header 'msg))

(defmacro is-message (thing)
  `(is-plist-with-header ,thing 'msg))

(defmacro new-declaration ()
  `(new-plist-with-header 'dcl))

(defmacro is-declaration (thing)
  `(is-plist-with-header ,thing 'dcl))

(defmacro get-content-slot (content &rest slots)
  `(get-package-slot ,content ,@slots))
```

```
(defmacro set-content-slot (content value &rest slots)
  `(set-package-slot ,content ,value ,@slots))

(defmacro content-body-as-list (content)
  `(cdr ,content))
```

A.7 File discourse.lisp

```
;;;
;;; RCS: $Header: /u6/grads/hans/Kqml/Code/discourse.lisp,v 1.1 92/05/15 15:19:01 hans Exp Locker: hans $
;;;

(in-package "KQML")


;; General Discourse:

(defclass discourse ()
  ((request-content-language
    :documentation
    "The content language used in requests"
    :initform "interlingua"
    )
   (reply-content-language
    :documentation
    "The content language used in replies. If this slot is unbound it
    defaults to the value of request-content-language."
    )
   (local-content-language
    :documentation
    "The content language used in the local application"
    )
   (local-address
    :documentation
    "Address of the local application engaged in this discourse"
    )
   (remote-address
    :documentation
    "Address of the remote application engaged in this discourse"
    )
   (stream
    :documentation
    "The bidirectional stream via which requester and provider communicate"
    )
   (control
    :documentation
    "T if the local application is in control of the discourse, NIL otherwise.
     Issuing a request to a provider sets this to NIL. Once a success reply
     for that request has come in, the control will be reset to T."
    )
   (history
    :documentation
    "The history of packages exchanged by the requester and the provider
     in this particular discourse (a list with the most recent package as
     its first element)"
    :initform nil
    )
   )
  (:documentation
   "Instances of class DISCOURSE contain all the necessary state information
to execute one-to-one discourse between a particular requester and the local
provider. Subclasses can provide specific methods to deal with language
```

```
specific translations of requests into commands understood by the local
application.")
  )

(defmethod slot-unbound (class
                         (instance discourse)
                         (slot (eql 'reply-content-language))
                         )
  "Provides default access to REQUEST-CONTENT-LANGUAGE if
REPLY-CONTENT-LANGUAGE is unbound."
  (declare (ignore class))
  (slot-value instance 'request-content-language))


;; SNePS Discourse:

(defclass sneps-discourse (discourse)
  ((request-content-language :initform "snepslin")
   (local-content-language :initform "sneps")
   (variable-mapping
    :initform (make-mapping)
    :documentation
    "Mapping between request content language variables and SNePS
variable nodes"
    )
   (discourse-context
    :initform (intern (symbol-name (gentemp "KQML-DISCOURSE-CONTEXT"))
                      'snepsul)
    :documentation
    "The name of the SNePS context used for the current SNePS discourse"
    )
   )
  (:documentation
   "Instances of class SNEPS-DISCOURSE contain all the necessary state
information for a discourse with a SNePS system.")
  )


;; Definition of performatives:
;;
;; For every possible KQML message expression there exists a ``message
;; method'' or performative with (almost) identical argument list that
;; handles a message of that :TYPE.

(defvar *performatives* nil
  "List of KQML performatives")

(defun is-performative (name)
  (and (member name *performatives*)
       (fboundp name)))

(defmacro defperformative (name &rest definition)
  `(progn
    (pushnew ',name *performatives*)
    (defmethod ,name ,@definition)))
```

A.8 File kqml-server.lisp

```
;;;
;;; RCS: $Header: /u6/grads/hans/Kqml/Code/kqml-server.lisp,v 1.2 92/05/16 16:56:09 hans Exp Locker: hans $
;;;

(in-package "KQML")


;; General TCP communication stuff:
;;
;; Tokens transmitted accross a TCP/IP stream are divided into 3 classes:
;;
;; 1) symbols:  Symbols are used to communicate simple commands or states,
;;              or to identify services
;; 2) strings:  Strings are used to transmit error messages
;; 3) packages: Packages (i.e., lists for which is-package is T) are used
;;              to transmit KQML messages
;;
;; These tokens are read with the READ function and classified by their
;; datatype. Everything else that does not belong to any of the three
;; classes is considered an illegal token.

(defun is-eof (input)
  (eq input :eof))

(defun is-bye (input)
  (eq input :bye))
(defun signal-bye (stream)
  (format stream ":bye~%")
  (force-output stream))

(defun is-ok (input)
  (eq input :ok))
(defun signal-ok (stream)
  (format stream ":ok~%")
  (force-output stream))

(defun is-error-message (input)
  (stringp input))
(defun signal-error (stream &rest format-args)
  (format stream "~s" (apply #'format (cons nil format-args))))


;; Establishing the service:

(defvar *local-address* (machine-instance))
(defvar *kqml-service* tcp::*lisp-server-port*)

(defun kqml-service-id (&optional application-id)
  "Creates a service id for a KQML server depending on APPLICATION.
Using an APPLICATION id additionally to the internet address to identify
a server allows us to communicate between processes that run on the same
machine, and/or to have multiple KQML servers running in a single process."
  (intern (format nil "KQML~@[-~a~]" application-id) 'keyword))
```

```
;; To establish a connection between a client and a KQML server the following
;; two steps have to be taken:
;;
;; 1) the server has to enable the service, for example, if the server runs
;;    on hadar.cs.buffalo.edu and has App-01 as an application id, then the
;;    service can be enabled with
;;
;;       (enable-kqml-service "App-01" 'sneps-discourse)
;;
;; 2) Once the service has been enabled, the client can establish a discourse
;;    with the server by doing the following:
;;
;;       (setq d (open-kqml-discourse "App-01@hadar.cs.buffalo.edu"
;;                                    'sneps-discourse)
;;
;; This discourse instance d can then be used to carry out a conversation
;; with the server.


;; The server side:
;;
(defun enable-kqml-service (&optional application-id
                                      (discourse-type 'discourse))
  "Starts the tcp::inetd server and enables a DISCOURSE-TYPE KQML
service identified by APPLICATION-ID. The tcp::service property of the
kqml-service-id will be bound to the kqml-server function, which will
be started by the tcp::inetd server if an appropriate service request
is received from a client."
  (tcp:start-server *kqml-service*)
  (let* ((service-id (kqml-service-id application-id))
         (service-function-name (gensym "kqml-server-")))
    ;; tcp::inetd expects a symbol as the service function
    (setf (symbol-function service-function-name)
          #'(lambda (stream remote-host)
              (kqml-server stream remote-host application-id discourse-type)))
    (setf (get service-id 'tcp::service)
          service-function-name)))

(defun disable-kqml-service (&optional application-id)
  "Disables a particular KQML service by setting the tcp::service property
to NIL."
  (let ((service-id (kqml-service-id application-id)))
    (setf (get service-id 'tcp::service) nil)))


;; The client side:
;;
(defun connect-to-kqml-server (address)
  "Opens and returns a stream to a KQML server running at the host
specified by ADDRESS. ADDRESS can be either a normal Internet address, or
additionally specify an application as in app01l@address. If no application
was specified it tries to connect to a service called :KQML. Otherwise it
tries to connect to a service called :KQML-<application id>."
  (if (symbolp address)
      (setf address (symbol-name address)))
  (check-type address string)
```

```
  (let* ((atsign-index (position #\@ address))
         (host-address
          (cond (atsign-index (subseq address (1+ atsign-index)))
                (t address)))
         (application-id
          (and atsign-index
               (subseq address 0 atsign-index)))
         (service-id
          (kqml-service-id application-id))
         (tcp-stream (tcp:connect-to-service host-address *kqml-service*))
         acknowledgment)
    (when tcp-stream
      ;; If we successfully opened an Internet stream, and the remote
      ;; server has enabled a KQML service with the same service-id,...
      (format tcp-stream "~s~%" service-id)
      ;; ...then the function kqml-server will be started in a separate
      ;; process, and if it starts up ok it will acknowledge the proper
      ;; establishment of the connection.
      (setq acknowledgment (read tcp-stream nil :eof)))
    (cond ((is-ok acknowledgment) tcp-stream)
          ((is-error-message acknowledgment)
           (warn "Could not connect to ~a: ~a"
                 host-address acknowledgment)
           nil)
          (t (warn "Could not connect to ~a" host-address)
             nil))
    ))

(defun open-kqml-discourse (address &optional (discourse-type 'discourse))
  "Opens a connection to a KQML server at ADDRESS, and if successful
returns a discourse instance of DISCOURSE-TYPE with properly
initialized stream and address slots."
  (let ((stream (connect-to-kqml-server address))
        discourse)
    (when stream
      (setq discourse (make-instance discourse-type))
      (setf (slot-value discourse 'stream) stream)
      (setf (slot-value discourse 'local-address) *local-address*)
      (setf (slot-value discourse 'remote-address) address)
      discourse)))

(defun close-kqml-discourse (discourse)
  "Terminates a KQML discourse by closing the stream specified in DISCOURSE."
  (let ((stream (slot-value discourse 'stream)))
    (signal-bye stream)
    (close stream)))


;; Sending and receiving packages:

(defun make-package-id (address)
  "Generates a unique package ID (a string) composed from the ADDRESS of
the sender and a time stamp."
  (format nil "~a ~d" address (get-internal-real-time)))

(defun dummy-id ()
```

```
  "Generates a new dummy request identifier"
  (symbol-name (gentemp "RQ-")))

(defun send-kqml-package (package discourse)
  "Deliver a KQML PACKAGE to a stream specified in DISCOURSE."
  ;; First store the package in the history of the sender (so we do
  ;; remember that we sent it)
  (setf (slot-value discourse 'history)
        (cons package (slot-value discourse 'history)))
  ;; Now, write it into the discourse stream
  (format (slot-value discourse 'stream) "~S~%" package))

(defun find-package-in-history (id discourse)
  "Tries to find a package with ID in the history of DISCOURSE, and returns
it if it was found."
  (find-if #'(lambda (pkg)
               (equal (get-package-slot pkg :id) id))
           (slot-value discourse 'history)))

(defun send-request (content discourse)
  "Send a request package with a CONTENT expression to a provider
specified in DISCOURSE."
  (let* ((provider-address (slot-value discourse 'remote-address))
         (local-address (slot-value discourse 'local-address))
         (request-package (new-package)))
    (set-package-slot request-package content :content)
    (set-package-slot request-package 'sync :comm)
    (set-package-slot request-package (make-package-id local-address) :id)
    (set-package-slot request-package provider-address :to)
    (set-package-slot request-package local-address :from)
    (send-kqml-package request-package discourse)))

(defun send-reply (content discourse)
  "Send a reply package with a CONTENT expression to a requester specified
by a package in the history of DISCOURSE whose :id slot is equal to the
:request-id of CONTENT. If no such request exists nothing will be sent."
  (let* ((request-id (get-content-slot content :request-id))
         (request-package
          (find-package-in-history request-id discourse))
         (requester-address (get-package-slot request-package :from))
         (local-address (slot-value discourse 'local-address))
         (reply-package (new-package)))
    ;; Don't send unsolicited replies:
    (when request-package
      (set-package-slot reply-package content :content)
      (set-package-slot reply-package 'sync :comm)
      (set-package-slot reply-package (make-package-id local-address) :id)
      (set-package-slot reply-package requester-address :to)
      (set-package-slot reply-package local-address :from)
      (send-kqml-package reply-package discourse))))


(defun interpret-package (package discourse)
  "Assumes that PACKAGE is a proper package, checks whether it contains
a defined performative in its content slot, and if so constructs a
proper argument list from the content of the package and applies the
```

```
appropriate performative (or message) method to them."
        (let* ((content (get-package-slot package :content))
               (performative (get-content-slot content :type))
               arguments)
          (cond ((and (is-message content)
                      (is-performative (get-content-slot content :type)))
                 (setf (slot-value discourse 'history)
                       (cons package (slot-value discourse 'history)))
                 (setq arguments
                       (append (list
                                discourse :id (get-package-slot package :id))
                               (content-body-as-list content)))
                 (when-debug
                  (format t "~a: ~s~2%" performative arguments))
                 (apply performative arguments)
                 )
                (t (signal-error (slot-value discourse 'stream)
                                 "Illegal package: ~a" package)
                   ))))

(defun interpret-tcp-stream-input (input discourse)
  "Function used by requester and provider to interpret a single INPUT
token read from the stream specified in DISCOURSE. Handles eof, closing,
error messages and interpretation of KQML packages."
  (let ((remote-host (or (slot-value discourse 'remote-address)
                         "foreign host"))
        (stream (slot-value discourse 'stream)))
    (when-debug
     (format t "~&TCP stream input: ")
     (write input :pretty t :escape t)
     (format t "~2%"))
    (cond ((is-eof input)
           (when-debug (format t "~&EOF reached on KQML stream."))
           (close stream))
          ((is-bye input)
           (when-debug
            (format t "~&Bye, KQML stream closed by ~a" remote-host))
           (close stream))
          ((is-ok input))
          ((is-error-message input)
           (warn "[~a] reported by ~a" input remote-host))
          ((is-package input)
           (interpret-package input discourse))
          (t (signal-error
              stream "Illegal input: ~a" input)
             ))))


;; The KQML server:
;;
(defun kqml-server (stream remote-host application-id discourse-type)
  "Server function invoked in a separate process by the tcp::inetd
function. It creates and initializes a discourse instance of
DISCOURSE-TYPE and then reads and interprets incoming KQML packages
from STREAM sent by REMOTE-HOST until the connection gets closed.
APPLICATION-ID in conjunction with *local-address* generate the from
```

```
address for packages sent back to the requester."
  ;; This is the outermost error handler on the server side that handles
  ;; everything that is not already handled by specific KQML error handlers
  (handler-case
   (let ((discourse (make-instance discourse-type))
         (*package* (find-package 'kqml))
         (process *current-process*)
         input)
     (setf (slot-value discourse 'stream) stream)
     (setf (slot-value discourse 'remote-address) remote-host)
     (setf (slot-value discourse 'local-address)
           (cond (application-id
                  (format nil "~a@~a" application-id *local-address*))
                 (t *local-address*)))
     ;; Do some process renaming
     (when (eq (process-initial-function process) 'tcp::inetd)
       (setf (process-name process) "KQML server"))
     ;; Acknowledge proper connection
     (signal-ok stream)
     (loop
      ;; Reading should problaby be error-handled individually to catch
      ;; cases such as undefined # reader macros etc.
      (setq input (read stream () :eof))
      (setf (slot-value discourse 'control) t)
      (interpret-tcp-stream-input input discourse)
      (when (or (is-eof input)
                (is-bye input))
        (return))
      ))
    ;; Cleanup if we crashed, so requester won't wait forever
    (error (condition)
           (ignore-errors
            ;; Try to inform requester about what happened
            (signal-error
             stream "KQML server died: ~a" condition))
           (close stream)
           (ignore-errors
            (warn "KQML server died: ~a" condition)))))

(defun send-request-and-interpret-replies (content discourse)
  "The client side of the communication. It sends a package with
a CONTENT expression as a request to the KQML server specified by
DISCOURSE, and then reads and interprets replies until a success-reply
was read or some error has occured. The evaluation of a success-reply
will set the discourse control to T which will terminate this function."
  (send-request content discourse)
  (let* ((*package* (find-package 'kqml))
         (stream (slot-value discourse 'stream))
         input)
    (setf (slot-value discourse 'control) nil)
    (loop
     (setq input (read stream () :eof))
     (interpret-tcp-stream-input input discourse)
     (when (or (slot-value discourse 'control)
               (not (is-package input)))
       (return)))))
```

A.9 File inference-control.lisp

```
;;;
;;; RCS: $Header: /u6/grads/hans/Kqml/Code/inference-control.lisp,v 1.1 92/05/15 15:29:09 hans Exp Locker: hans
;;;

(in-package "KQML")

;; The functions below implement control for suspending inference.
;; Suspending inference is different from standard inference in that
;; the inference process is not carried out until no more information
;; can be generated, but rather suspended when some criterion is met
;; (e.g., a certain # of answers got generated). At the point of
;; suspension enough state information of the inference engine is saved
;; so that the suspended inference can be continued right from where
;; it was stopped, just as if it had not been interrupted at all.

;; Deal with general suspending inference:

(defun suspend-inference (multi:*NAME*)
  "Special process function that can be scheduled at the front of the
high priority queue. It saves all the necessary multi status such that
inference can be properly continued after it was suspended by this
function."
  (declare (ignore multi:*NAME*))
  (setf (get 'lastinfer 'event-queues)
        (list multi::*high-priority-queue*
              multi::*low-priority-queue*))
  (setf (get 'lastinfer 'user-process) snip::*USER-PROCESS*)
  (multi:clear-all-queues))

(setf (get 'suspend-inference 'multi::lregs%) '(multi:*NAME*))

(defvar *suspend-process* (multi:new 'suspend-inference)
  "Process used to suspend forward inference")

;; Deal with suspending BACKWARD inference:
;;
;; Suspending backward inference can naturally be handled with the
;; #-of-answers argument supplied to deduce. Once that many answers
;; have been generated, the USER process calls the funcion
;; snip::suspend-inference and returns the newly derived answers.
;;
;; What we have to do is to modify this functions so that it saves the
;; multi state in order to allow us proper inference continuation.

(defun snip::suspend-inference ()
  (suspend-inference 'ignore))

;; Deal with suspending FORWARD inference:
;;
;; Interruption of forward inference is more tricky, because there we
;; have no concept of an answer to a query, rather newly added nodes
;; depending on the grinding of the inference engine.  What we want is
;; to suspend inference every time a new node is added to the network
;; during the process of forward inference. How this is done is
```

```
;; explained below.
;; A new command ADD-SUSPENDINGLY starts suspending forward inference.
;; This is an almost identical copy of ADD, with the only difference
;; that it binds the variables *suspending-forward-inference-p* and
;; *previously-added-nodes* which tell other functions that they deal
;; with suspending forward inference, and what previous results have
;; been inferred so far.

(defvar *suspending-forward-inference-p* nil)
(defvar *previously-added-nodes* (sneps:new.ns))

(defmacro add-suspendingly (&rest snd)
  `(let* ((sneps:crntct (sneps:processcontextdescr ',snd))
          (snip:crntctname sneps:crntct)
          (*suspending-forward-inference-p* t)
          (*previously-added-nodes* (sneps:new.ns)))
     (declare (special sneps:crntct snip:crntctname))
     (values (snip::add* (sneps::nseval (cons 'sneps:assert ',snd)))
             snip:crntctname)))

(setf (get 'add-suspendingly 'sneps::=command) t)
(setf (get 'add-suspendingly 'sneps::=topcommand) t)

;; The trick is to find the right spot where forward inference should be
;; suspended to report some new result. After careful examination it turns
;; out that there are only three places where something gets added to the
;; variable snip:*ADDED-NODES*:
;;    In snip:process-one-report.non-rule,
;;       snip:process-one-instance-report.rule and
;;       snip:try-to-use-introduction-conclusion
;; All of these functions call broadcast-one-report immediately after they
;; updated the variable *ADDED-NODES*. Hence, changing broadcast-one-report
;; to check whether inference suspension is appropriate seems to be a good
;; way to go.

;; Here is a handcoded ADVISE for broadcast-one-report:
(defvar *broadcast-one-report* (symbol-function 'snip::broadcast-one-report)
  "Original definition of snip::broadcast-one-report")

;; Now redefine it:
(defun snip::broadcast-one-report (&rest args)
  (declare (special snip::*added-nodes*))
  (let ((result (apply *broadcast-one-report* args)))
    (when (and *suspending-forward-inference-p*
               ;; Check whether we have some newly added nodes (the initially
               ;; added node will also always suspend inference):
               (not (sneps:isnew.ns
                     (sneps:compl.ns snip::*added-nodes*
                                     *previously-added-nodes*))))
      ;; Scheduling a suspension process at the front of the high priority
      ;; queue will make sure that the immediately next thing after the
      ;; termination of the current process will be inference suspension.
      ;; This has to be done in this way, because there will be more processes
      ;; scheduled by the current process after this point, hence we cannot
      ;; save the state right here, rather we have to save the state right
      ;; after the current process terminated.
```

```
      (dequeue::insert-front *suspend-process* multi::*high-priority-queue*)
      )
    result))

;; Deal with inference continuation:
;;
;; continue-suspended-inference relies on the assumption that whenever
;; inference is suspended a non-empty set of answers will be returned,
;; unless all inference possiblities have been exhausted.

(defun continue-suspended-inference (&optional (mode 'backward))
  "Continues processing of previously saved multi queues. This function
will always return a result as long as more answers could possibly get
generated by another inference continuation. Once no new result could
get generated and the queues are empty NIL will be returned. This can
be used to determine whether all inference possibilities have been
exhausted. Depending on MODE the proper variables for collecting new
results are bound."
  (declare (special snip::*added-nodes*))
  (let ((*suspending-forward-inference-p* (eq mode 'forward))
        (*previously-added-nodes* snip::*added-nodes*)
        (previous-answers
         (case mode
           (forward snip::*added-nodes*)
           (backward snip::*deduction-results*)))
        (user-process (get 'lastinfer 'user-process))
        (event-queues (get 'lastinfer 'event-queues))
        )
    (apply #'multi::multip event-queues)
    (values (sneps::compl.ns
             (case mode
               (forward snip::*added-nodes*)
               (backward snip::*deduction-results*))
             previous-answers)
            (multi:regfetch user-process 'snip::*context-name*))
    ))

(defvar *current-inference-mode* 'backward)

(defun continue-inference ()
  "SNePSUL command that continues suspended inferenence of any kind.
Instead of having two different continuation commands for the two
inference directions, this command figures out what kind of inference
preceded it and continues accordingly."
  (let ((mode (case (first (sneps:value.sv 'sneps::lastcommand))
                ((snip:add add-suspendingly)
                 (setq *current-inference-mode* 'forward))
                (snip:deduce
                 (setq *current-inference-mode* 'backward))
                (t *current-inference-mode*))))
  (continue-suspended-inference mode)))

(setf (get 'continue-inference 'sneps::=command) t)
(setf (get 'continue-inference 'sneps::=topcommand) t)
```

A.10 File kqml-performatives.lisp

```
;;;
;;; RCS: $Header: /u6/grads/hans/Kqml/Code/kqml-performatives.lisp,v 1.7 92/05/16 18:51:31 hans Exp Locker: hans
;;;

(in-package "KQML")


(defmethod valid-request-language ((discourse discourse) language)
  "Returns nonNIL if LANGUAGE is a valid content language for requests."
  (stringp language))

(defmethod valid-reply-language ((discourse discourse) language)
  "Returns nonNIL if LANGUAGE is a valid content language for replies."
  (stringp language))


;; General (application independent) message methods:

(defperformative declare-content-languages ((discourse discourse)
                                            &key
                                            (request-content-language nil)
                                            (reply-content-language nil)
                                            (id (dummy-id)))
  "Handles KQML messages of that :TYPE"
  (with-kqml-errors
      (when request-content-language
        (cond ((valid-request-language discourse request-content-language)
               (setf (slot-value discourse 'request-content-language)
                     request-content-language))
              (t (kqml-error
                  (format nil "Don't know how to translate ~a into ~a"
                          request-content-language
                          (slot-value discourse 'local-content-language))))))
    (when reply-content-language
      (cond ((valid-reply-language discourse reply-content-language)
             (setf (slot-value discourse 'reply-content-language)
                   reply-content-language))
            (t (kqml-error
                (format nil "Don't know how to translate ~a into ~a"
                        (slot-value discourse 'local-content-language)
                        reply-content-language)))))
    (send-success-reply discourse :value 'success :request-id id)))


;; Deal with local execution of KQML performatives that should
;; not send a reply of any sort back to the requester:

(defvar *execute-locally* nil
  "Flag that gets bound to T if a certain performative should not generate
replies from a local usage.")

(defmacro execute-locally (&body body)
  `(let ((*execute-locally* t))
     ,@body))
```

```
(defmacro unless-only-locally (&body body)
  `(unless *execute-locally*
     ,@body))


(defmethod send-success-reply ((discourse discourse)
                               &key
                               (value nil)
                               (request-id nil)
                               (explanation nil))
  "Sends KQML message of that :TYPE"
  (unless-only-locally
   (when-debug
    (case value
      (failure
       (format t "~&Request ~a could not be handled successfully, because:~
                ~%  ~a" request-id explanation)
       :failure)
      (success
       (when explanation
         (format t "~&Request ~a handled successfully.~
                  ~%  ~a" request-id explanation))
       :success)))
   (let ((reply (new-message)))
     (set-content-slot reply explanation :explanation)
     (set-content-slot reply request-id :request-id)
     (set-content-slot reply value :value)
     (set-content-slot reply 'success-reply :type)
     (send-reply reply discourse))))

(defperformative success-reply ((discourse discourse)
                                &key
                                (value nil)
                                (request-id nil)
                                (explanation nil)
                                (id (dummy-id)))
  "Handles KQML messages of that :TYPE"
  (declare (ignore id))
  (case value
    (failure
     (format t "~&Request ~a could not be handled successfully, because:~
         ~      ~%  ~a" request-id explanation)
     :failure)
    (success
     (when explanation
       (format t "~&Request ~a handled successfully.~
                  ~%  ~a" request-id explanation))
     :success))
  (when (find-package-in-history request-id discourse)
    (setf (slot-value discourse 'control) T)))

(defmethod send-content-reply ((discourse discourse)
                               &key
                               (request-id nil)
                               (reply-number 1)
```

```
                              (reply-content-language nil)
                              (content nil))
  "Sends KQML messages of that :TYPE"
  (unless-only-locally
   (when-debug
    (format t "~&Reply #~d to request ~a:~%" reply-number request-id))
   (let ((translated-sentence-set
          (mapcar #'(lambda (sentence)
                      (translate-reply-content
                       discourse
                       :content sentence
                       :content-language reply-content-language))
                  content)))
     (when translated-sentence-set
       (when-debug
        (format t "  { ~a~{~%    ~a~} }~2%"
                (car translated-sentence-set)
                (cdr translated-sentence-set)))
       (let ((reply (new-message)))
         (set-content-slot
          reply (format nil "~s" translated-sentence-set) :content)
         (set-content-slot reply reply-number :reply-number)
         (set-content-slot reply request-id :request-id)
         (set-content-slot reply 'content-reply :type)
         (send-reply reply discourse)))
     )))

(defun listify-string-sentence-list (string-sentence-list)
  "Some messages take a list of content sentences which is stored as a
string. To be able to map over such a list, it has to be converted
into a list of strings (one for each sentence in the list)."
  (mapcar #'(lambda (sentence)
              (format nil "~s" sentence))
          (read-from-string string-sentence-list)))

(defperformative content-reply ((discourse discourse)
                                &key
                                (request-id nil)
                                (reply-number 1)
                                (content nil)
                                (id (dummy-id)))
  "Handles KQML messages of that :TYPE"
  ;; This should be done more elegantly:
  (setq content (listify-string-sentence-list content))
  (when-debug
   (format t "~&Reply #~d to request ~a:~%" reply-number request-id)
   (format t "  { ~a~{~%    ~a~} }~2%"
           (car content)
           (cdr content)))
  (dolist (sentence content)
    (execute-locally
     (assert discourse
             :request-content-language (slot-value discourse
                                                   'reply-content-language)
             :content sentence
             id (dummy-id)))))
```

```
;; SNePS specific message methods:

(defmethod valid-request-language ((discourse sneps-discourse) language)
  (and (stringp language)
       (or (string-equal language "snepsul")
           (string-equal language "snepslin"))))

(defmethod valid-reply-language ((discourse sneps-discourse) language)
  (and (stringp language)
       (or (string-equal language "snepsul")
           (string-equal language "snepslin"))))

(defmacro sneps-execute (&body body)
  "Executes SNePS code supplied in BODY. This macro could actually bundle
up the code in a closure and queue it to be executed in a SNePS process.
So far it just binds the error condition to catch internal errors in SNePS."
  `(handler-bind ((error #'(lambda (condition)
                             (invoke-toplevel-restart
                              'sneps-system-error
                              condition))))
     (let ((*package* (find-package 'snepsul)))
       ,@body)))

(defun modify-sneps-context (context-name translated-content mode)
  "Modifies the context with name CONTEXT-NAME according to MODE (one of
:SET, :ADD or :REMOVE). TRANSLATED-CONTENT is a list that describes a
pattern which can be use as the rest of a FIND command to find nodes
that should be added/removed to the current discourse context."
  (let ((matched-assertions
         (case translated-content
           (empty ())
           (all #!(*assertions))
           ;; Find every assertion in the whole network that matches the
           ;; supplied pattern. For :remove just look in the discourse ctxt
           (t (let ((snip:*infertrace* nil)
                    all-matches current-matches)
                ;; To be able to use just one kind of sentence-pattern
                ;; I use deduce 0 here instead of findassert. This makes it
                ;; a bit more difficult to get all matches, because it only
                ;; returns one answer at a time, hence we have to loop with
                ;; continue deduction until we found all of them:
                (setq all-matches
                      (sneps-execute
                       #3!((clear-infer)
                           (deduce 0 ~@translated-content
                                   :context ~(case mode
                                               ((:set :add) 'all-hyps)
                                               (:remove context-name))))))
                (loop
                 (setq current-matches
                       (sneps-execute
                        #3!((continue-inference))))
                 (cond ((sneps:isnew.ns current-matches)
                        (return all-matches))
```

```
                    (t (setq all-matches (sneps:union.ns
                                          all-matches
                                          current-matches)))))
              )))))
    (case mode
      (:set (sneps-execute
             #3!((set-context ~matched-assertions ~context-name))))
      (:add (sneps-execute
             #3!((add-to-context ~matched-assertions ~context-name))))
      (:remove (sneps-execute
                #3!((remove-from-context
                     ~matched-assertions ~context-name))))
      )))

(defmethod translate-request-content ((discourse sneps-discourse)
                                      &key content content-language)
  (let ((request-content-language
         (cond ((and content-language
                     (valid-request-language discourse content-language))
                content-language)
               (content-language
                (kqml-error
                 (format nil "Don't know how to translate ~a into ~a"
                         content-language
                         (slot-value discourse 'local-content-language))))
               (t (slot-value discourse 'request-content-language)))))

    (cond (;; For SNePSUL input just read it into the SNePSUL package with
           ;; the SNePS readtable
           (string-equal request-content-language "snepsul")
           (let* ((*package* (find-package 'snepsul))
                  (*readtable* sneps::*sneps-readtable*))
             (read-from-string content)))

          ;; For SNePSLin read the string into the SNePSUL package and then
          ;; translate it into a SNePSUL form. Implicitly define new relations
          ((string-equal request-content-language "snepslin")
           (let* ((*package* (find-package 'snepsul)))
             (multiple-value-bind (snepsul-expression relations)
                 (snepslin-to-snepsul (read-from-string content) discourse)
               ;; Implicitly define new relations referenced in this sentence
               (dolist (rel relations)
                 (unless (sneps:is.r rel) (sneps:new.r rel)))
               snepsul-expression)))

          ;; For a valid language which we don't know return nilllll
          (t nil))))

(defmethod translate-reply-content ((discourse sneps-discourse)
                                    &key content content-language)
  (let ((reply-content-language
         (cond ((and content-language
                     (valid-reply-language discourse content-language))
                content-language)
               (content-language
                (kqml-error
```

```
                (format nil "Don't know how to translate ~a into ~a"
                        (slot-value discourse 'local-content-language)
                        content-language)))
              (t (slot-value discourse 'reply-content-language)))))

    (cond ((string-equal reply-content-language "snepsul")
           (list content (sneps:node-fcableset content)))
          ((string-equal reply-content-language "snepslin")
           (sneps-to-snepslin content discourse))
          ;; For a valid language which we don't know return nillllll
          (t nil))))

(defperformative set-discourse-context ((discourse sneps-discourse)
                                        &key
                                        (request-content-language nil)
                                        (content 'all)
                                        (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (with-kqml-errors
      (modify-sneps-context
       (slot-value discourse 'discourse-context)
       (case content
         ((all empty) content)
         (t (translate-request-content
             discourse
             :content content
             :content-language request-content-language)))
       :SET)
    (send-success-reply
       discourse
       :value 'success
       :request-id id)
    ))

(defperformative add-to-discourse-context ((discourse sneps-discourse)
                                           &key
                                           (request-content-language nil)
                                           (content 'all)
                                           (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (with-kqml-errors
      (modify-sneps-context
       (slot-value discourse 'discourse-context)
       (case content
         ((all empty) content)
         (t (translate-request-content
             discourse
             :content content
             :content-language request-content-language)))
       :ADD)
    (send-success-reply
       discourse
       :value 'success
       :request-id id)
    ))
```

```
(defperformative remove-from-discourse-context ((discourse sneps-discourse)
                                                &key
                                                (request-content-language nil)
                                                (content 'empty)
                                                (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (with-kqml-errors
      (modify-sneps-context
       (slot-value discourse 'discourse-context)
       (case content
         ((all empty) content)
         (t (translate-request-content
             discourse
             :content content
             :content-language request-content-language)))
       :REMOVE)
    (send-success-reply
       discourse
       :value 'success
       :request-id id)
    ))


;; Now I am in trouble: In order to handle messages of that kind I need
;; to define relations (and paths). There are various ways to do this:
;;    - Define them implictly by scanning the snepsul sentence for
;;      undefined relations (does not work for paths)
;;    - Define a new KQML performative that handles definitions
;;    - Use a hack (an "interpret-literally" performative)

(defperformative assert ((discourse sneps-discourse)
                         &key
                         (request-content-language nil)
                         (content nil)
                         (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (with-kqml-errors
      (let (;; Do this outside the sneps-execute because the translation
            ;; might fail, which is a KQML error and not a SNePS error
            (translated-content
             (translate-request-content
              discourse
              :content content
              :content-language request-content-language)))
        (sneps-execute
         #!((assert ~@translated-content
                    :context ~(slot-value discourse 'discourse-context))))
        (send-success-reply
         discourse
         :value 'success
         :request-id id)
        )))

;; New KQML performative that takes a sentence and executes it as an
;; application command without any interpretation (hack which handles
;; everything that is not handled by any proper performative):
```

```
(defperformative interpret-literally ((discourse sneps-discourse)
                                      &key
                                      (content nil)
                                      (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (with-kqml-errors
      (let ((translated-content
             (translate-request-content
              discourse
              :content content
              :content-language "snepsul"))
            (old-default-context #!(*defaultct)))
        (sneps-execute
         #!((set-default-context ~(slot-value discourse 'discourse-context))
            ~translated-content
            (set-default-context ~old-default-context)))
        (send-success-reply
         discourse
         :value 'success
         :request-id id)
        )))

(defperformative assign-truth-value ((discourse sneps-discourse)
                                     &key
                                     (request-content-language nil)
                                     (truth-value nil)
                                     (content nil)
                                     (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (declare (ignore request-content-language truth-value content))
  (with-kqml-errors
      (send-success-reply
       discourse
       :value 'failure
       :request-id id
       :explanation
       (format nil
               "SNePS does not deal with truth values. The assertion status~
            ~%  of nodes can be changed with assert and ~
                remove-from-discourse-context."))))

;; Issues:
;;   - clear-infer?? follow-up questions?
;;   - truth values: Believed, conceived, negation believed?
;;

;; INFER-ANSWERS (by disobeying all  rules of software engineering)
;; cramps forward/backward inference, proper answer reporting, handling
;; of worklevel etc. all in one function, so that the same workhorse can
;; be used by all 3 KQML query methods to do their thing.

(defmethod infer-answers ((discourse sneps-discourse)
                          &key
                          (request-content-language nil)
                          (reply-content-language nil)
                          (worklevel 'minimal)
```

```
                        (how-many 1)
                        (report-mode 'continuous)
                        (truth-values 'any)
                        (topics nil)
                        (inference-mode 'backward)
                        (id (dummy-id)))
  "Does the actual work for topical, forward and backward queries. It
tries to infer answers related to the TOPICS (in the order given) by
either forward or backward inference depending on INFERENCE-MODE. If
INFERENCE-MODE is forward, TOPICS are considerd to be seeds from which
new information should be derived by forward inference. In the case of
backward inference, TOPICS are considerd to be queries.  Each answer
gets derived according to WORK-LEVEL and TRUTH-VALUES. Results are
reported according to HOW-MANY, REPORT-MODE and request ID."
  (declare (ignore truth-values))
  (let ((reported-answers (sneps:new.ns))
        (reply-number 0))
    (dolist (content topics)
      ;; First check all the parameters for proper values!!
      (let* ((translated-content
              (translate-request-content
               discourse
               :content content
               :content-language request-content-language))
             (current-answers
              (sneps-execute
               (let ((snip:*infertrace* nil))
                 (case inference-mode
                   (backward
                    (sneps-execute
                     #3!(;; Currently, assume that every question is
                         ;; independent from previous ones, hence all
                         ;; node activations should get removed before
                         ;; new inference is done:
                         (clear-infer)
                         (deduce ~(case worklevel (minimal 0) (maximal 1))
                                 ~@translated-content
                                 :context ~(slot-value
                                            discourse 'discourse-context)
                                 ))))
                   (forward
                    (sneps-execute
                     #3!((clear-infer)
                         (add-suspendingly
                                 ~@translated-content
                                 :context ~(slot-value
                                            discourse 'discourse-context)
                                 ))))))))
             (new-answers
              (sneps:compl.ns current-answers reported-answers))
             )
        (loop
         (when-debug
          (format t "~&CUR: ~a NEW: ~a REP: ~a~2%"
                  current-answers new-answers reported-answers))
```

```
        ;; If we have enough new answers report them
        (when (or (>= (sneps:cardinality.ns new-answers) how-many)
                  (sneps:isnew.ns current-answers))

          (when new-answers
            (send-content-reply discourse
                                :request-id id
                                :reply-number (incf reply-number)
                                :content new-answers
                                :content-language reply-content-language)
            (setq reported-answers
                  (sneps:union.ns new-answers reported-answers))
            (setq new-answers (sneps:new.ns)))

          ;; Figure out whether we are done
          (cond ((or (sneps:isnew.ns current-answers) ; no (more) answers
                     ;; For minimal worklevel we use 0 in the deduce for
                     ;; which a continuation of the inference doesn't make
                     ;; sense, hence in this case we alway return after the
                     ;; first set of results was reported
                     (eq worklevel 'minimal))
                 (return))
                (t;; otherwise, check whether we have to suspend inference
                 (case report-mode
                   (suspend;; do proper suspending, HOW???
                    (return))
                   (continuous nil)))))

        ;; Try to find more answers
        (setq current-answers
              (let ((snip:*infertrace* nil))
                (sneps-execute
                 #3!((continue-inference)))))
        (setq new-answers
              (sneps:compl.ns (sneps:union.ns current-answers new-answers)
                              reported-answers))
        )))))

(defperformative query-sentence-status ((discourse sneps-discourse)
                                        &key
                                        (request-content-language nil)
                                        (reply-content-language nil)
                                        (worklevel 'minimal)
                                        (how-many 1)
                                        (report-mode 'continuous)
                                        (truth-values 'any)
                                        (content nil)
                                        (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (with-kqml-errors
      ;; First check all the parameters for proper values!!
      (infer-answers discourse
                     :request-content-language request-content-language
                     :reply-content-language reply-content-language
                     :worklevel worklevel
                     :how-many how-many
```

```
                    :report-mode report-mode
                    :truth-values truth-values
                    :id id
                    :inference-mode 'backward
                    :topics (list content))
    (send-success-reply
     discourse
     :value 'success
     :request-id id)
    ))

;; TOPICAL QUERIES:
;;
;; The trick is to determine what questions to ask in order to derive
;; information that's related to a topic described by a content sentecne
;; or phrase (currently, I can't handle phrases - atomic things, that is).
;;
;; The approach taken is to take the topic, translate it into a SNePS
;; node and match it against the network to generate a set of matching
;; nodes. Every matching node has a (maybe empty) substitution
;; associated with it. We then take the matching node (unless it is a
;; variable node) and all its dominating nodes and apply the substitution
;; to each of them. The resulting set of nodes will be added to a set
;; of "related questions" generated by doing the same thing for all
;; the other matching nodes.
;;
;; The so generated set of related questions will then be passed one
;; by one to deduce to find out about possible answers.

(defun dominating-nodes (node)
  "Returns the set of nodes dominating NODE"
  (let ((dominating (sneps:new.ns)))
    (sneps:do.fcs (rel ns (sneps::up.fcs node) dominating)
      (sneps:do.ns (n ns)
        (setq dominating
              (sneps:union.ns (sneps:insert.ns n dominating)
                              (dominating-nodes n)))
        ))))

;; To do:
;; Augment snepslin-to-snepsul so that it can handle atoms and sentences.
;; Write some predicate methods that can check whether something is a
;; sentence or not....
(defperformative query-about-topic ((discourse sneps-discourse)
                                    &key
                                    (request-content-language nil)
                                    (reply-content-language nil)
                                    (worklevel 'minimal)
                                    (how-many 1)
                                    (report-mode 'continuous)
                                    (truth-values 'any)
                                    (content nil)
                                    (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (declare (ignore truth-values))
  (with-kqml-errors
```

```
   ;; First check all the parameters for proper values!!
   (let* ((translated-content
           (translate-request-content
            discourse
            :content content
            :content-language request-content-language))
          ;; if the topic was a sentence we have to build the
          ;;
          (topic-node
           (sneps-execute
            ;; build returns a singleton set
            #2!((build ~@translated-content))))
          (related-questions topic-node)
          )
     (match:do.supmatchingset (supmatching
                               (match:match-in-context
                                (sneps:choose.ns topic-node )
                                (sneps:value.sv
                                 (slot-value
                                  discourse 'discourse-context))))
       (let ((matching-node (match:tnode.supmatching supmatching))
             (substitution (match:target-sub.supmatching supmatching))
             )
         (unless (sneps:isvar.n matching-node)
           (sneps:do.ns (node (sneps:insert.ns
                               matching-node
                               (dominating-nodes matching-node)))
             (setq related-questions
                   (sneps:insert.ns
                    (match::applysubst node substitution)
                    related-questions))
             ))))
     (infer-answers discourse
                    :request-content-language "snepslin"
                    :reply-content-language reply-content-language
                    :worklevel worklevel
                    :how-many how-many
                    :report-mode report-mode
                    :truth-values truth-values
                    :id id
                    :inference-mode 'backward
                    ;; Because infer-answers cannot handle actual
                    ;; SNePS nodes, we translate them into a snepslin
                    ;; description first (not very efficient, but it works)
                    :topics (mapcar #'(lambda (related-question)
                                        (format nil "~a"
                                                (sneps-to-snepslin
                                                 related-question
                                                 discourse)))
                                    related-questions))
     )
   (send-success-reply
    discourse
    :value 'success
    :request-id id)
   ))
```

```
;; FORWARD INFERENCE:
;;
(defperformative assert-and-infer ((discourse sneps-discourse)
                                   &key
                                   (request-content-language nil)
                                   (reply-content-language nil)
                                   (worklevel 'minimal)
                                   (how-many 1)
                                   (report-mode 'continuous)
                                   (truth-values 'any)
                                   (assertion-mode 'actual)
                                   (content nil)
                                   (id (dummy-id)))
  "Handles KQML messages of that :TYPE that are directed to SNePS"
  (with-kqml-errors
      (let ((sentence-list (listify-string-sentence-list content)))

        ;; First, assert all the sentences in the current context:
        (dolist (sentence sentence-list)
          (execute-locally
           (assert discourse
                   :request-content-language request-content-language
                   :reply-content-language reply-content-language
                   :content sentence
                   :id (dummy-id))))

        ;; Now do the inference:
        (infer-answers discourse
                       :request-content-language request-content-language
                       :reply-content-language reply-content-language
                       :worklevel worklevel
                       :how-many how-many
                       :report-mode report-mode
                       :truth-values truth-values
                       :id id
                       :inference-mode 'forward
                       :topics sentence-list)

        ;; If we were only asked hypothetically, remove all sentences and
        ;; their dependent results from the current context:
        (when (eq assertion-mode 'hypothetical)
          (dolist (sentence sentence-list)
            (execute-locally
             (remove-from-discourse-context
              discourse
              :request-content-language request-content-language
              :content sentence
              :id (dummy-id)))))

        (send-success-reply
         discourse
         :value 'success
         :request-id id)
        )))
```

A.11 File remote-sneps.lisp

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; Package: SNEPS; Base: 10 -*-

(in-package "KQML")

(rcs-info "$Header: /u6/grads/hans/Kqml/Code/remote-sneps.lisp,v 1.1 92/05/15 15:16:46 hans Exp Locker: hans $"


;; A set of experimental SNePS top-level commands that deal with
;; the communication with a remote knowledge base via KQML messages.

;; Discourse variables:
;;
;; The SNePSUL variable default-discourse holds the name of the
;; default discourse (in a list), its default value is
;; (default-default-discourse)
;; A discourse name is itself a SNePSUL variable whose value is
;; an actual discourse object (as a list)

(defmacro open-sneps-discourse (address &optional discourse-name)
  "Opens a connection to ADDRESS and if successful binds the
SNePSUL variable DISCOURSE-NAME to it. If no name was supplied
the value of DEFAULT-DISCOURSE will be used (which will be initialized
to DEFAULT-DEFAULT-DISCOURSE if it was empty). The supplied discourse
name will be made the default discourse. Initializes the remote
context to empty, and declares SNePSLin as the discourse language."
  `(let ((dis (open-kqml-discourse ,address 'sneps-discourse))
         (disname (or ',discourse-name
                      (car #!(*default-discourse))
                      (car #!((= (^ 'default-default-discourse)
                                 default-discourse)))))
         (context #!(*defaultct))
         request)
     (when dis
       #!((= (^ '~dis) ~disname))
       #!((= (^ '~disname) default-discourse))
       (setf (slot-value dis 'discourse-context) context)
       ;; declare content languages
       (setq request (new-message))
       (set-content-slot request 'declare-content-languages :type)
       (set-content-slot request "snepslin" :request-content-language)
       (send-request-and-interpret-replies request dis)
       ;; initialize discourse context
       (setq request (new-message))
       (set-content-slot request 'set-discourse-context :type)
       (set-content-slot request 'empty :content)
       (send-request-and-interpret-replies request dis)
       (values disname context))))

(setf (get 'open-sneps-discourse 'sneps::=command) t)
(setf (get 'open-sneps-discourse 'sneps::=topcommand) t)

(defmacro close-sneps-discourse (&optional discourse-name)
  "Closes the discourse with name DISCOURSE-NAME."
  `(let* ((disname (or ',discourse-name
```

```
                    (car #!(*default-discourse))))
          (discourse (car #!(*~disname)))
          (context #!(*defaultct)))
    (when (typep discourse 'sneps-discourse)
      (close-kqml-discourse discourse)
      #!((= () ~disname))
      (values disname context))))

(setf (get 'close-sneps-discourse 'sneps::=command) t)
(setf (get 'close-sneps-discourse 'sneps::=topcommand) t)


;; Macro for the definition of "remote commands"
;;
(defmacro def-snepsul-remote (command &rest definition)
  "Defines a SNePSUL command with name remote-COMMAND whose properties
are the same as the SNePSUL COMMAND. Does proper import/export too."
  (let ((remote-command
         (intern (format nil "REMOTE-~a" (symbol-name command)) 'kqml)))
    `(progn
      (export ',remote-command 'kqml)
      (shadowing-import ',remote-command 'snepsul)
      (setf (symbol-plist ',remote-command)
       (symbol-plist (intern (symbol-name ',command) 'snepsul)))
      (defmacro ,remote-command ,@definition))))

;; Syntax of remote command calls:
;;
;; A call to a remote command is similar to a call of a normal
;; SNePSUL command:
;;
;;  (remote-command ....node specification... :context ct :discourse d)
;;
;; The context and discourse arguments are optional, they default to
;; *defaultct and *default-discourse. Results from the remote execution
;; of the remote-command will be asserted in the specified context.

(defun parse-remote-cmd-arguments (arguments)
  "Takes a list of ARGUMENTS, extracts the optional context and discourse
specifications, builds a node from the remaining args, and returns the
node, a context name and a discourse object."
  (setq arguments (copy-list arguments))
  (let* ((context-key-pos
          (position :context arguments :from-end t))
         (context-name
          (and context-key-pos
               (nth (1+ context-key-pos) arguments)))
         (discourse-key-pos
          (position :discourse arguments :from-end t))
         (discourse-name
          (and discourse-key-pos
               (nth (1+ discourse-key-pos) arguments)))
         (dummy (gensym)))
    (cond (context-name
           (setf (nth (1+ context-key-pos) arguments) dummy)
           (setf (nth context-key-pos arguments) dummy))
```

```
        (t (setq context-name #!(*defaultct))))
    (cond (discourse-name
           (setf (nth (1+ discourse-key-pos) arguments) dummy)
           (setf (nth discourse-key-pos arguments) dummy))
          (t (setq discourse-name (car #!(*default-discourse)))))
    (unless (sneps::context-p #!(*~context-name))
      (sneps:sneps-error
       (format nil "~a is not the name of a context" context-name)
       'remote-sneps
       'parse-remote-cmd-arguments))
    (unless (typep (car #!(*~discourse-name)) 'discourse)
      (sneps:sneps-error
       (format nil "~a is not the name of a discourse" discourse-name)
       'remote-sneps
       'parse-remote-cmd-arguments))
    (values (car #2!((build ~@(remove dummy arguments))))
            context-name
            (car #!(*~discourse-name)))))


;; Some examples of how SNePSUL remote commands can be written
;; using the KQML performatives:

(def-snepsul-remote assert (&rest node-specification)
  "Asserts a node defined by NODE-SPECIFICATION in the remote discourse
context"
  `(multiple-value-bind (node context discourse)
    (parse-remote-cmd-arguments ',node-specification)
    (setf (slot-value discourse 'discourse-context) context)
    (let ((request (new-message)))
     (set-content-slot request
                       (format nil "~s" (sneps-to-snepslin node discourse))
                       :content)
     (set-content-slot request 'assert :type)
     (send-request-and-interpret-replies request discourse))))

(def-snepsul-remote findassert (&rest node-specification)
  "Tries to find asserted nodes that match NODE-SPECIFICATION in
the remote discourse context. Answers will be asserted in the local
discourse contxt (this is different from the standard find-assert
semantics, because find normally just searches but does not build anything."
  `(multiple-value-bind (node context discourse)
    (parse-remote-cmd-arguments ',node-specification)
    (setf (slot-value discourse 'discourse-context) context)
    (let ((request (new-message)))
     (set-content-slot request
                       (format nil "~s" (sneps-to-snepslin node discourse))
                       :content)
     (set-content-slot request 'continuous :report-mode)
     (set-content-slot request 1 :how-many)
     (set-content-slot request 'minimal :worklevel)
     (set-content-slot request 'query-sentence-status :type)
     ;; Use kqml::old-nodes so we won't interfere with user variables
     #!((= *nodes ~~'old-nodes))
     (send-request-and-interpret-replies request discourse)
     ;; Nodes asserted by this command that have not been in the
```

```
    ;; network previously will be returned.
    (values #!((- *nodes *~~'old-nodes))
            context))))

(def-snepsul-remote deduce (&rest node-specification)
  "Tries to find answers for NODE-SPECICATION in the remote discourse
context by way of backward inference. Answers found will be asserted
as hypotheses rather than derived nodes as done by the standard deduce."
  `(multiple-value-bind (node context discourse)
    (parse-remote-cmd-arguments ',node-specification)
    (setf (slot-value discourse 'discourse-context) context)
    (let ((request (new-message)))
     (set-content-slot request
                       (format nil "~s" (sneps-to-snepslin node discourse))
                       :content)
     (set-content-slot request 'continuous :report-mode)
     (set-content-slot request 1 :how-many)
     (set-content-slot request 'maximal :worklevel)
     (set-content-slot request 'query-sentence-status :type)
     #!((= *nodes ~~'old-nodes))
     (send-request-and-interpret-replies request discourse)
     (values #!((- *nodes *~~'old-nodes))
             context))))

(def-snepsul-remote add (&rest node-specification)
  "Tries to find answers for NODE-SPECICATION in the remote discourse
context by way of forward inference. Answers found will be asserted
as hypotheses rather than derived nodes as done by the standard add."
  `(multiple-value-bind (node context discourse)
    (parse-remote-cmd-arguments ',node-specification)
    (setf (slot-value discourse 'discourse-context) context)
    (let ((request (new-message)))
     (set-content-slot request
                       (format nil "(~s)" (sneps-to-snepslin node discourse))
                       :content)
     (set-content-slot request 'actual :assertion-mode)
     (set-content-slot request 'continuous :report-mode)
     (set-content-slot request 1 :how-many)
     (set-content-slot request 'maximal :worklevel)
     (set-content-slot request 'assert-and-infer :type)
     #!((= *nodes ~~'old-nodes))
     (send-request-and-interpret-replies request discourse)
     (values #!((- *nodes *~~'old-nodes))
             context))))
```