Knowledge Modelling in WebOnto and OCML

A User Guide

John Domingue, Enrico Motta, and

Oscar Corcho Garcia

Version 2.4

Table of contents

Introduction

The WebOnto User Guide

Knowledge Modelling in OCML

INTRODUCTION

This document is composed of three parts. The first part describes WebOnto, a tool providing web-based visualisation, browsing and editing support for developing and maintaining ontologies and knowledge models specified in OCML. The description is user-oriented, in the sense that it is meant to provide guidance to a user, rather than to describe the tool from a scholarly perspective. The second part of the document describes OCML, an operational knowledge modelling language, which provides the underlying representation for the ontologies and knowledge bases which can be developed using WebOnto. This second part is a revised version of chapter 4 of (Motta, 1999). The third part (Appendix 1), gives more details about the interpreters and reasoning facilities provided by OCML.

This document is available from the WebOnto home page at http://kmi.open.ac.uk/projects/webonto.

Chapter 1

The WebOnto User Guide

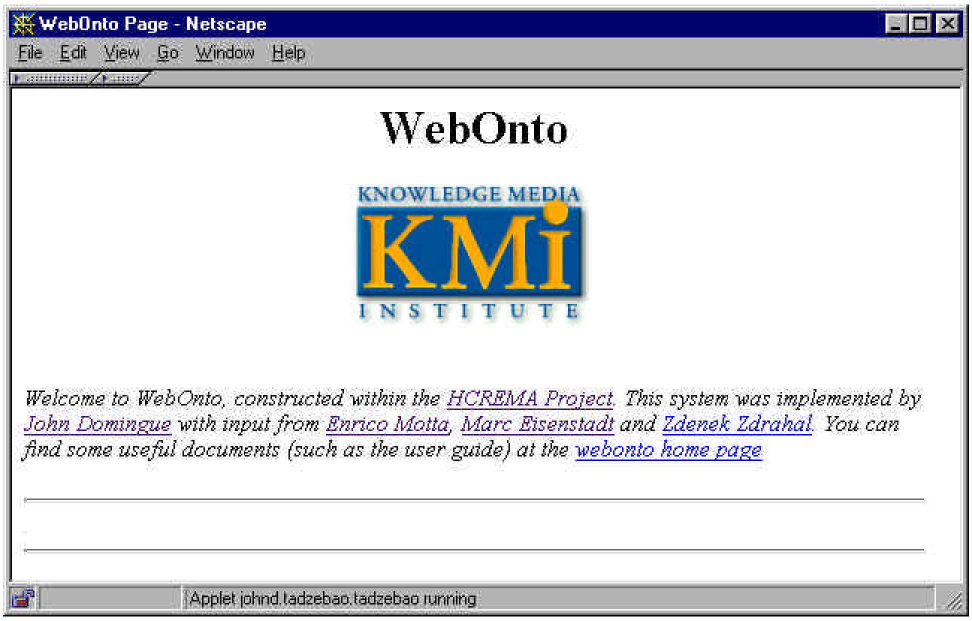
John Domingue

1 OVERVIEW

You can start WebOnto by going to http://webonto.open.ac.uk/. This page has the appearance of figure 1. The page contains an invisible applet from which the classes for the WebOnto applet are loaded. After 20 seconds the applet frames shown in figures 2 and 3 should appear. We call these the main WebOnto window and the fine-grained view window.

Figure 2 shows a screen snapshot of the WebOnto client browsing the knowledge base called kmi-planet-kb-drafts-220799. The main window at the top has five areas, which are:

- The title bar – this shows the name of the knowledge model currently being browsed and the type of the components currently displayed in the Item List. OCML distinguishes between four types of knowledge models: domain ontologies, problem solving methods, task ontologies and knowledge bases. The title in figure 2 indicates that we are currently viewing the classes in kmi-planet-kb-drafts-220799 knowledge base.
- The menu bar – we describe the menu items in the following section.
- The tool bar – the tool bar icons are described in section 2.8.
- The graphical display area. Graphical representations of selected objects from the Item list can be displayed here.



WebOnto

Welcome to WebOnto, constructed within the HCREMA Project. This system was implemented by John Domingue with input from Enrico Motta, Marc Eisenstadt and Zdenek Zdrahal. You can find some useful documents (such as the user guide) at the webonto home page

Applet johnd.tadzebao.tadzebao running

Figure 1. The initial WebOnto page. The java classes are loaded from this page.

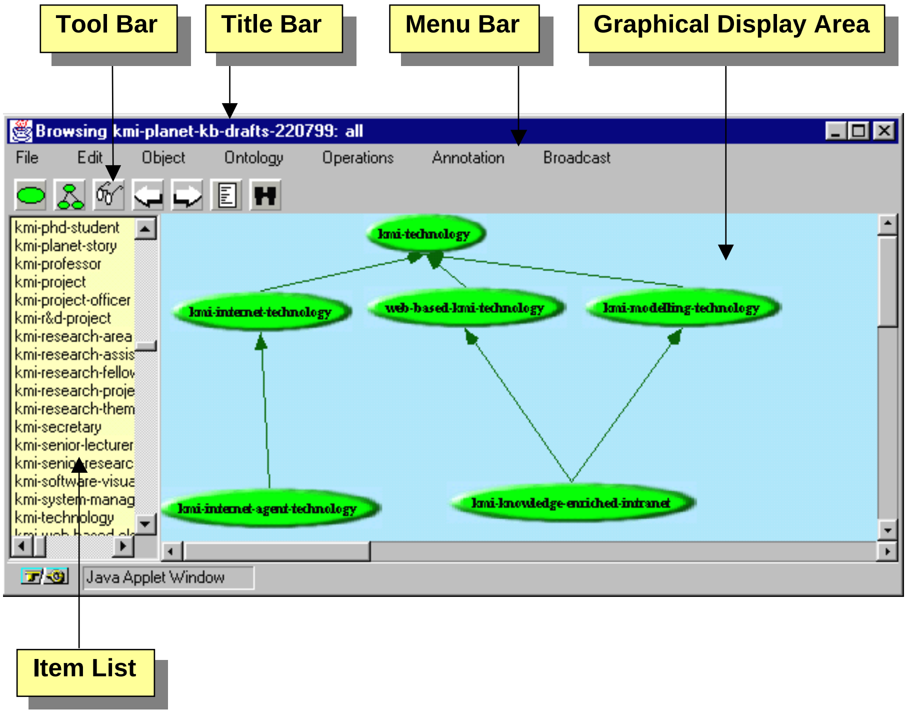
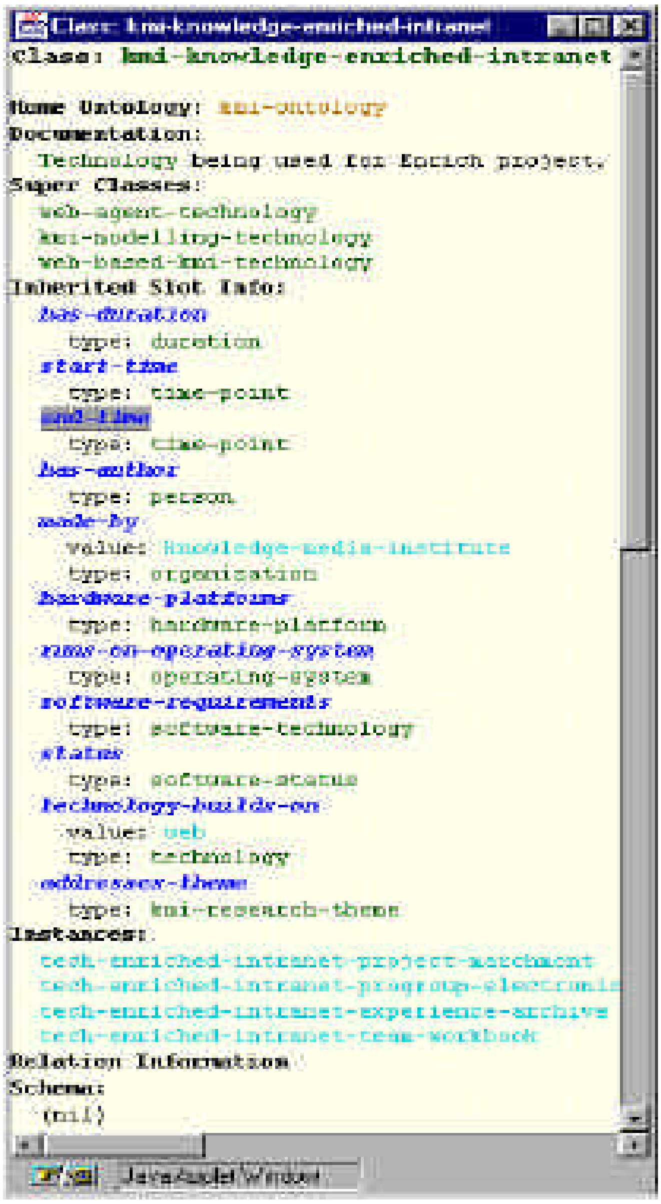


Figure 2. An annotated screen snapshot of WebOnto in browsing mode depicting the applet components.



```
Class: kmi-knowledge-enriched-intranet

Home Ontology: kmi-ontology
Documentation:
  Technology being used for Enrich project.
Super Classes:
  web-agent-technology
  kmi-modelling-technology
  web-based-kmi-technology
Inherited Slot Info:
  has-duration
    type: duration
  start-time
    type: time-point
  end-time
    type: time-point
  has-author
    type: person
  made-by
    value: knowledge-media-institute
    type: organization
  hardware-platforms
    type: hardware-platform
  runs-on-operating-system
    type: operating-system
  software-requirements
    type: software-technology
  status
    type: software-status
  technology-builds-on
    value: web
    type: technology
  addresses-theme
    type: kmi-research-theme
Instances:
  tech-enriched-intranet-project-merchant
  tech-enriched-intranet-progroup-electronic
  tech-enriched-intranet-experience-archive
  tech-enriched-intranet-team-workbook
Relation Information
Schema:
  (nil)
```

Figure 3. A snapshot of the fine-grained view window currently showing the class `kmi-knowledge-enriched-intranet`**.**

Figure 3 shows the fine-grained view window. Messages from the server and the internal descriptions of OCML components (displayed using the icon) are displayed here. Any text displayed in the fine-grained view window is automatically parsed and any word that corresponds to an OCML item is displayed in a specific colour. The colour of a word indicates the OCML type. The current colour coding is:

- Class – green,
- Instance - turquoise,

- Relation – blue,
- Function – violet,
- Procedure – violet-red,
- Rule - maroon,
- Ontology – orange.

Highlighted items can be selected and operated on. In figure 3 the relation `end-time` is currently selected. Double clicking on an item has the same effect as selecting it and clicking on the 👓 (inspect) icon.

Items can be selected in the Item List, Graphical Display Area or the Fine-Grained View window. Selecting an item in one area deselects any selected items in the other areas (but see known bug 1 in section 6).

2 MENU AND TOOL BARS

We shall now describe the menu bar. The menu bar contains the following menu items:

2.1 File

- **Reload Ontology** - reload an ontology on the server. This need only be used when a set of files have been placed on the server by the server administrator. All the knowledge models are loaded when the server is started up. When import/export facilities are added to WebOnto this menu item is expected to be more heavily used.
- **Login –** login into WebOnto. Note that one can only edit, create or delete knowledge models when logged in. Contact j.b.domingue@open.ac.uk if you require a login name and password.
- **Logout** – logout of WebOnto.
- **Change Password –** change the currently logged in user’s password.
- **New Ontology –** create a new knowledge model. This menu item is described in section 3.
- **Delete Ontology –** delete a knowledge model. This is menu item described in section 3.
- **Edit Ontology Properties –** edit the properties of a knowledge model. The properties include the author of the knowledge model and the additional users who are allowed to edit the knowledge model. This menu item is described in section 3.
- **Save diagram** – save the currently displayed items in the display area to a file on the server.
- **Open diagram** – open a previously saved diagram.
- **Reinitialise OCML** - clear all the currently defined OCML structures and reload the entire library on the server. Only the root user can do this.

- **Display Source File –** display a one of the source files for the current knowledge model. The user is prompted with the names of the source files on the WebOnto server where the current knowledge model is stored. The selected file is displayed in a display source window.
- **Debug** - turn on an internal debugger that sends a trace of all operations to the Java console window.
- **Quit** - quit WebOnto.

2.2 Edit

- **Graphical Cut** - remove the currently selected object from the graphical display area.
- **Delete –** delete the currently selected item (only available in edit mode).
- **Edit Source** - edit the currently selected object (only available in edit mode).
- **View Source –** display the OCML source of the currently selected object.
- **New Class Instance –** creates an instance definition form (see section 5.2), enabling an instance of the selected class to be created (only available in edit mode).
- **Search for String –** search for OCML entities within the library which contain the specified string.
- **Clear Diagram** - clear the graphical area.
- **Clear Before Drawing –** if selected the graphical area is always cleared before drawing a new item.
- **Node Highlighting** - turn on node highlighting. This feature uses the semantics of OCML to highlight which components are allowed to be connected to each other. This prevents users from attempting to create links that have no defined semantics in OCML (only available in edit mode).
- **Edit Mode –** when selected WebOnto is placed in edit mode. See section 5.1 for a description of WebOnto's edit mode.

2.3 Object

- **Draw** - draw the current item in the graphical display area.
- **Draw Tree –** draw a tree, in the graphical display area, if appropriate for the currently selected item. Currently trees are drawn for classes, tasks, problem solving methods and knowledge models.
- **Inspect** - inspect the currently selected object, displaying the result in the fine-grained view window.
- **Direct Instances** - show the direct instances of the currently selected class in the fine-grained view window.
- **All Instances** - show all the instances of the currently selected class in the fine-grained view window.

- **Display Direct Instances** - show the direct instances of the currently selected class in the graphical-display area.
- **Display All Instances** - show all the instances of the currently selected class in the graphical-display area.
- **Draw Tree Vertically** – if selected trees are drawn vertically rather than horizontally.

2.4 Ontology

- **View Classes** - display all the classes in the current knowledge model in the Item list.
- **View Top_classes** - display the classes with no parents in the current knowledge model in the Item list.
- **View Relations** - display all the relations in the current knowledge model in the Item list.
- **View Functions** - display all the functions in the current knowledge model in the Item list.
- **View Procedures** - display all the procedures in the current knowledge model in the Item list.
- **View Rules** - display all the rules in the current knowledge model in the item list.
- **View Instances** - display all the instances in the current knowledge model in the item list.
- **View Tasks** - display all the tasks in the current knowledge model in the item list.
- **View problem_solving_methods** - display all the PSMs in the current knowledge model in the item list.
- **View All** – display all the OCML structures in the current knowledge model in the item list.
- **View Only Items in Current Ontology** – if selected OCML entities from inherited knowledge models are not displayed.
- **Include Base Ontology** – if selected OCML entities from the base ontology are displayed.
- **Change Number of Viewable Items** – set a limit on the number of items which can be viewed in the item list. This option facilitates the browsing of large knowledge models. The default value is 150.
- **Add View Filter** – this menu item allows the user to create a set of filters in order to filter viewable items in the item list. They can be combined using the operators *and* and *or*.
  - **Add a String Filter -** show only items containing the specified string.
  - **Add a Not String Filter -** show only items not containing the specified string.

  - **Add a Letter Range Filter -** show only items with their first letter in the specified range.
  - **Add a Predicate Filter -** show only items for which the given predicate is true. For example, typing the class name *person* would cause only instances of *person* to be displayed.
- **Delete Filter –** delete a defined previously filter.
- **Select Ontology –** reads in the OCML structures in the selected knowledge model and displays all the classes in the item list. It should be noted that this is subtly different from the View items above. ‘View’ selects between different OCML types within a knowledge model. This item allows the user to switch knowledge models. It is recommended that this item is used when changing knowledge models.
- **Unlock Ontology –** when a knowledge model is edited the WebOnto server locks it so that no-one else can edit it at the same time. Unfortunately, if WebOnto crashes whilst a knowledge model is being edited, the knowledge model can remain locked. This item can be used to unlock the knowledge model. Only the root user can do this.
- **Graph Entire Ontology Tree -** draw a graph of the inheritance hierarchy of all the knowledge models in library.

2.5 Operations

- **Tell** - prompt for an OCML expression to invoke with the OCML ‘tell’ operation.
- **Ask** - prompt for an OCML expression to invoke with the OCML ‘ask’ operation.
- **Set of All** - prompt for an OCML expression to invoke with the OCML ‘setofall’ operation.
- **Unassert** - prompt for an OCML expression to unassert in the current knowledge model.
- **Evaluate** - prompt for an OCML expression to invoke with the OCML ‘evaluate’ operation.

2.6 Annotations

- **Trail –** if selected dragging with the mouse button pressed draws a trail, that is, a freehand line.
- **Trail Colour** - choose the colour to trail in.
- **Clear** - clear the currently displayed trails and text annotations.

2.7 Broadcast

- **Broadcast –** when selected WebOnto ‘broadcasts’ all the user’s interface actions to WebOnto clients in receive mode.

- **Receive** – when selected all the broadcast interface actions are ‘received’ by the current WebOnto. Receiving WebOnto clients are in effect ‘slaves’ of the broadcasting WebOnto.

2.8 Tool Bar

Below the menu bar there is a tool bar containing the following icons:

- Draw the selected OCML component in the graphical display area.

- Create a graphical representation for the currently selected OCML component. If a class is selected a class hierarchy is drawn. In figure 2 above the class hierarchy for the `kmi-technology` class is displayed.

- Inspect the currently selected OCML component. The results are displayed in the fine-grained view window. In figure 3 above the `kmi-technology` class is currently being inspected.

- View the last item that was inspected. The fine-grained view window keeps a history of all the items inspected. This icon enables the user to move back through the history.

- View the next item in the fine-grained view window history.

- View the OCML source of selected item.

- Search for a string in the library of knowledge models.

- Edit the source of the selected item (only available in Edit Mode).

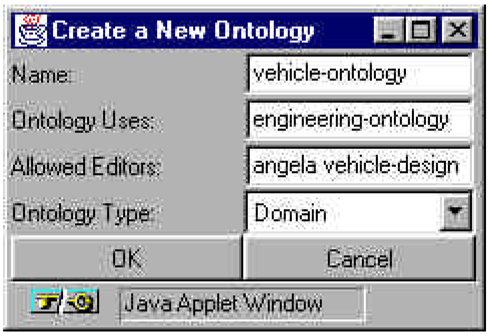
- Create new instance of the selected class (only available in Edit Mode).

3 CREATING AND DELETING KNOWLEDGE MODELS

We’ll now describe how WebOnto can be used to create, delete and edit the properties of a knowledge model. Figure 4 shows the dialog created when the menu item “New Ontology” (under the file menu) is chosen. The dialog is used to enter the four properties for the knowledge model, namely:

1. Name – the name of the knowledge model. In figure 4 an ontology called `vehicle-ontology` is being created.
2. Ontology Uses – the names of any knowledge models that this knowledge model should use (i.e. inherit from). The snapshot in figure 4 specifies that the vehicle-ontology will use the `engineering-ontology` ontology.
3. Allowed Editors – the names of any registered users or groups (each WebOnto user belongs to a group) who can edit the knowledge model. Figure 4 specifies that *angela* and any user belonging to the *vehicle-design* group will be allowed to edit the ontology.

4. Ontology Type – the type of the knowledge model, this can be one of: *domain*, *method*, *task* or *application/knowledge-base*.



Name: vehicle-ontology

Ontology Uses: engineering-ontology

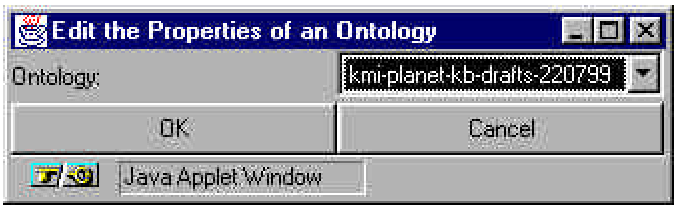
Allowed Editors: angela vehicle-design

Ontology Type: Domain

Figure 4. A screen snapshot of the 'Create New Ontology' dialog box.

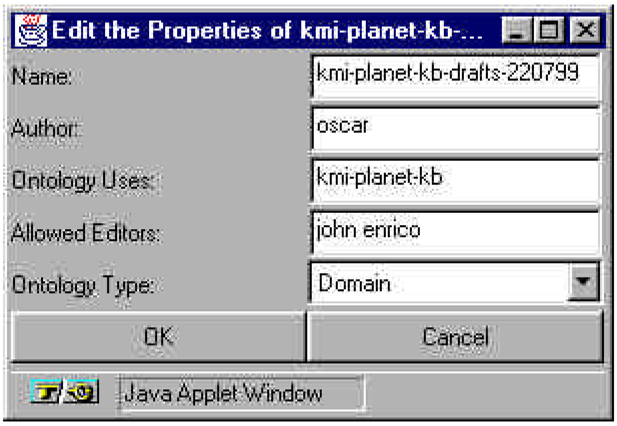
The "Delete Ontology" menu item under the file menu allows users to delete any knowledge models that they own. The dialogue box displayed for this menu item displays any knowledge models that the current WebOnto user owns (not the ones that s/he is just allowed to edit), except for the current knowledge model.

When a user selects the "Edit Ontology Properties" menu item under the file menu a dialog containing a menu of all the knowledge models owned by the user is displayed. A snapshot of this dialog is shown in figure 5. When the user selects the "OK" button the dialog shown in figure 6 is displayed. This dialog is similar to the dialog for creating new knowledge models. The only difference is that the dialog allows the author of the knowledge model to be changed.



Ontology: kmi-planet-kb-drafts-220799

Figure 5. A screen snapshot of the first 'Edit Ontology Properties' dialog box. The menu on the right contains all the knowledge models owned by the current user.



Edit the Properties of kmi-planet-kb-...

Name: kmi-planet-kb-drafts-220799

Author: oscar

Ontology Uses: kmi-planet-kb

Allowed Editors: john enrico

Ontology Type: Domain

Figure 6. A screen snapshot of the second 'Edit Ontology Properties' dialog box.

4 ITEM LIST

The item list displays items of the currently viewed knowledge model. The "Ontology" menu (see section 2.4) allows the user to control which items within the knowledge model are displayed. Using the "Ontology" menu the user can specify:

- That only items of a specified OCML type (classes, instances, relations, functions, procedures, rules, PSMs, tasks) are displayed.
- Whether OCML entities inherited from ancestor knowledge models are displayed.
- Whether OCML entities from the `base-ontology` are displayed.
- An numeric limit on the number of items displayed - this is useful when browsing very large ontologies,
- Filters which are used to cut down the number of items displayed. Again this is useful when browsing very large ontologies.

5 GRAPHICAL DISPLAY AREA.

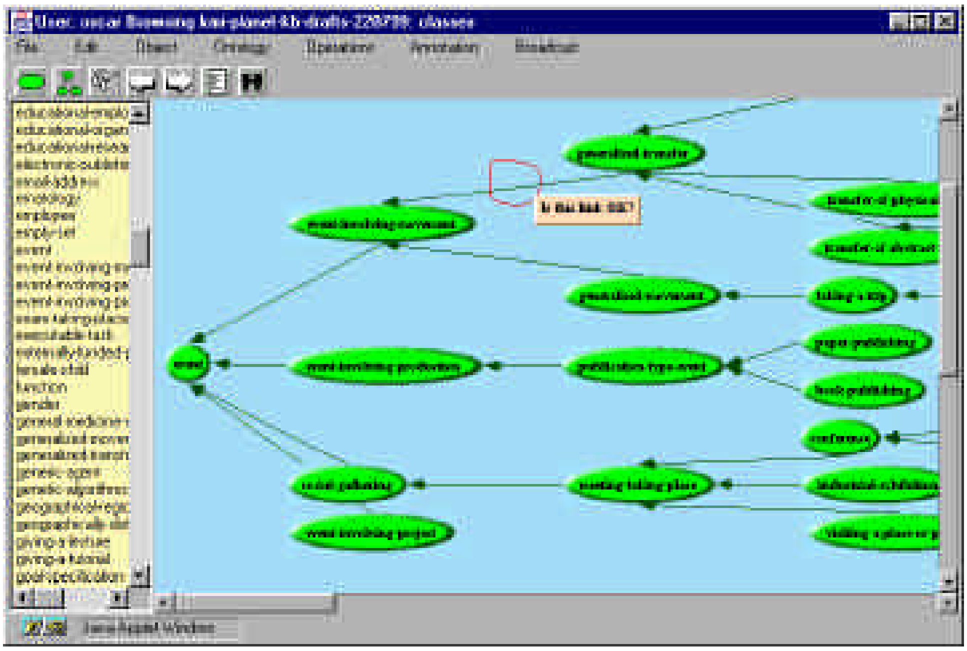


Figure 7. A screen snapshot showing a user annotating part of a class hierarchy in broadcast mode.

The graphical display area can be used for three different purposes:

- Knowledge model visualization –graphical representations of OCML entities can be drawn in the area (using the or icon).
- Synchronous communication – when WebOnto is in broadcast mode, interface actions are copied to all receiving WebOnto clients.
- Editing – OCML definitions can be created and edited using the graphical display area.

Figure 7 shows a user using the sketching facilities in WebOnto to communicate with a colleague whilst in broadcast mode. When the “Trail” menu item (under the “Annotation” menu (see section 2.6)) is selected dragging the mouse with the left button pressed draws a line in the selected “Trail Colour”. Clicking in the graphical area with the control button pressed creates a prompt for a text annotation (note that WebOnto must be in browsing mode). In figure 7 the user has drawn a red trail around a subclass link and then created a text annotation.

When the “Trail” button is not selected then dragging with the left mouse button draws a shaded area enabling multiple nodes to be selected. This is shown in figures 8 and 9 below. Single node selection is shown by four pimples – one at each corner of the node. Multiple node selection, as shown in figure 9, is shown by dark shading.

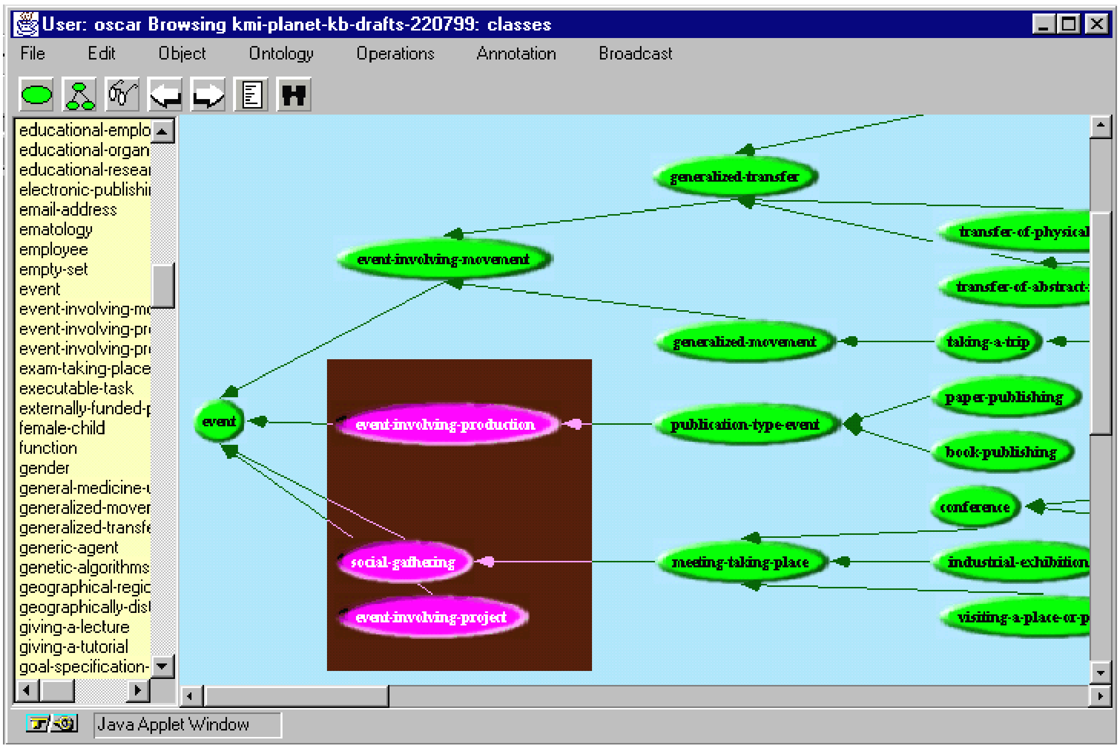


Figure 8. A screen snapshot showing a user selecting a number of nodes.

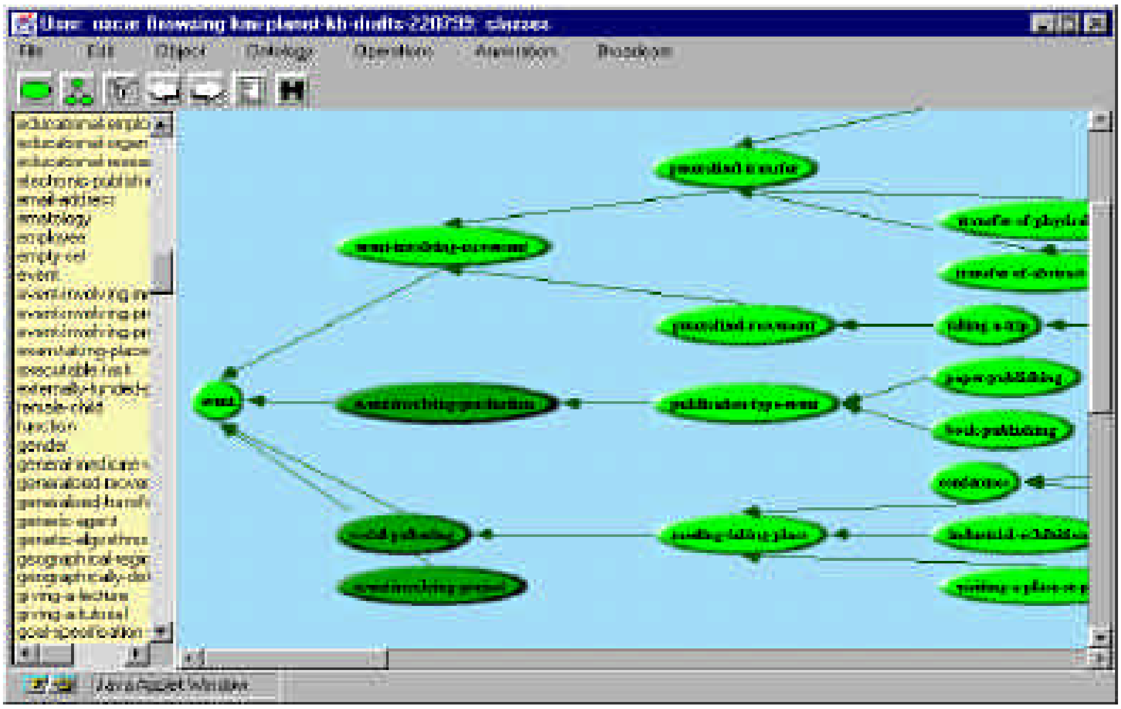


Figure 9. A screen snapshot showing 3 selected nodes. These nodes can be moved or cut in a single operation if desired.

5.1 Edit Mode

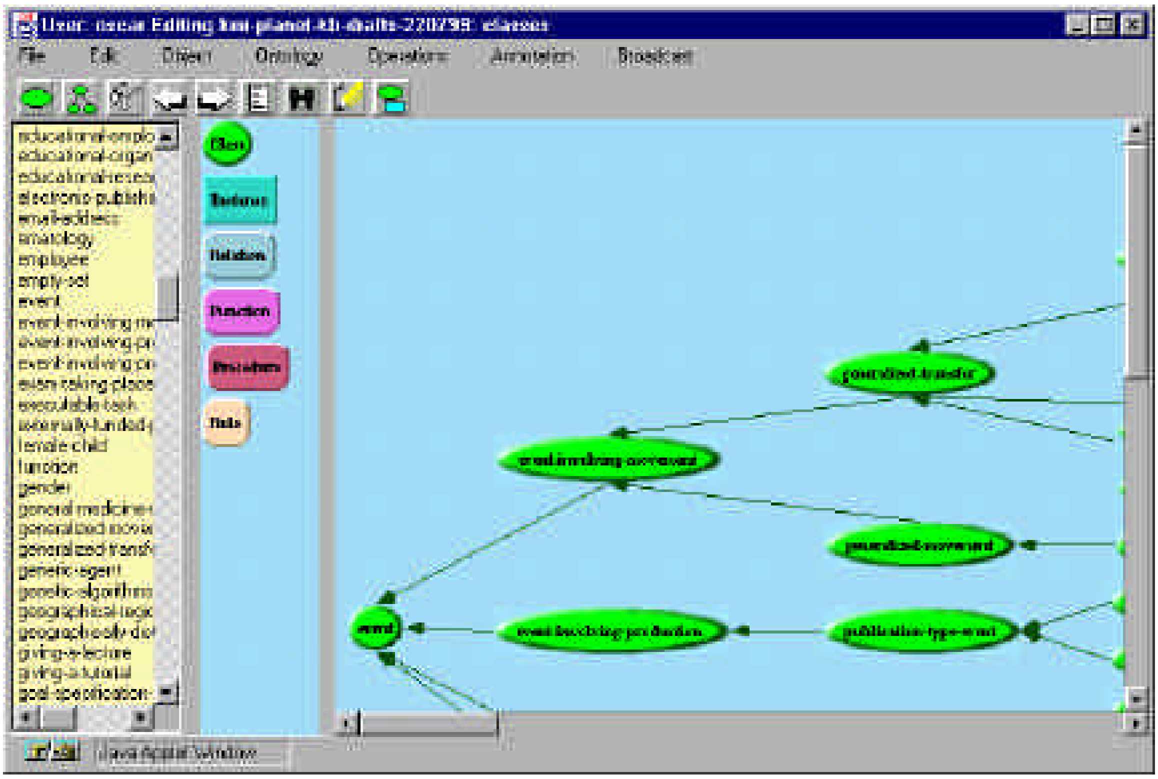


Figure 10. A screen snapshot showing how the appearance of WebOnto changes when it is in edit mode.

Figure 10 shows how the appearance of WebOnto changes when it is put into edit mode. Two new icons appear in the tool bar:

- Edit the currently selected object, and

- Create an instance of the currently selected class.

Additionally, a well from which new OCML definitions can be created is displayed. The graphical display area can be used to create OCML definitions by direct manipulation. WebOnto understands the semantics of OCML and prevents the user from creating semantically incorrect definitions in four ways:

1. *Node linking/highlighting* - Nodes can only be linked if this action is semantically correct. As the user moves the mouse around linkable nodes are highlighted. Node highlighting can be turned using the “Node Highlighting” menu item under the “Edit” menu.
2. *Automatic creation of templates and forms* - The manipulation of new nodes (e.g. linking them to existing nodes) results in the automatically insertion of source code. For example, if a new class `class1` is linked to an existing class `class2` the source

code `(defclass class1 (class2))` is written. The editing of instances is aided by the automatic creation of forms. These are described in more detail in section 5.2.

3. *Prompting for component parts* - in figure 15 the user has linked the class `academic` to the class `url`. When a user does this a selector (`works-`at in figure 15) is automatically created. The selector contains all the slots within the `academic` class, so the user has to merely select a slot rather than recall and type a slot's name.
4. *Semantic colour coding* - all the links drawn by the user between graphical icons are automatically colour coded depending on the type of link (e.g. class to subclass links are dark green, and class to instance links are blue).

5.2 Scenario

We shall describe the features available in WebOnto's edit mode using a short scenario. In the scenario below we describe how a user adds some classes and instances to the `tutorial-ontology` ontology. The user selects the ontology and initiates an editing session in using the following steps. The user:

1. Selects "Clear Diagram Before Drawing" under the "Edit" menu.
2. Selects the "Graph Entire Ontology Tree" menu item under the "Ontology" menu (see figure 11). Each knowledge model within the displayed tree is colour coded according to the knowledge model type:
    1. Beige – the `basic-ontology`,
    2. Light gold – a domain ontology,
    3. Light red – an application or knowledge base,
    4. Light blue – a problem solving method,
    5. Yellow – a task ontology.
3. Selects the node `tutorial-ontology` (see figure 11).
4. Selects the "Select Ontology" menu item under the "Ontology" menu.
5. Selects the "Edit Mode" menu item under the "Edit" menu.

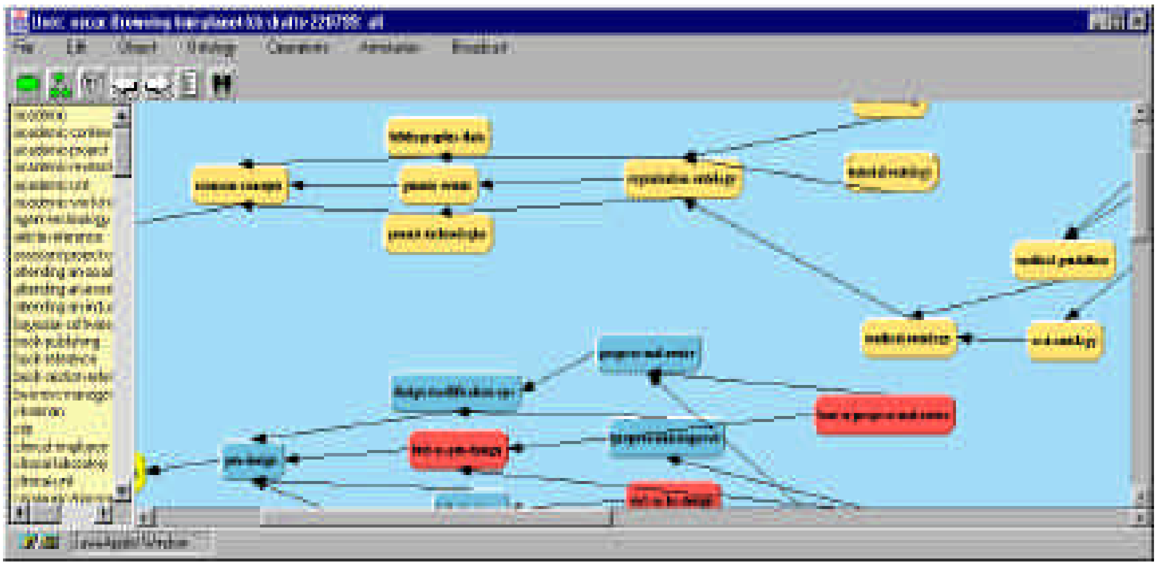


Figure 11. A screen snapshot of WebOnto displaying the structure of the knowledge models within the library. This can be obtained by selecting the "Graph Entire Ontology Tree" menu item under the "Ontology" menu. The colours indicate the knowledge model type.

The user defines a new class `senior_lecturer` which is a subclass of the class `academic` by using the following steps. Specifically, the user:

1. Selects the `academic` class in the item list" and clicks on the icon.
2. Drags the `Class` node from the well onto the graphical display area. Notice (see figure 12) that the node label for the new class `Class1` is displayed in a bold italic font because no definition for the node exists on the WebOnto server.
3. Clicks on the `academic` class node in the graphical display area with the mouse, whilst holding the control key down, and drags a line to the `Class1` node.
4. Selects the `Class1` node and clicks on the icon. This creates the edit window shown in figure 13.
5. Changes the name of the class in the edit window (see figure 13) to "senior_lecturer" and hits the "Done" button.

The user then specifies that the `has-web-address` slot of the `academic` class should be of type `url` by carrying out the following steps. The user:

1. Defines a new class `url` in the following manner. The user drags the `Class` node from the well onto the graphical display area, clicks on the icon, edits the code in the edit window changing the class name from "Class2" to "url" and then clicks on the "Done" button.
2. Clicks on the class `academic` with the mouse and drags to the class `url`. The user presses and holds down the "Shift" button and then releases the mouse button (see figure 14).

3. Clicks on the label “unset” and chooses “has-web-address” from the selector label (see figure 15). The definition of the `academic` class is automatically changed to:

```
(def-class academic (educational-employee)
  ((has-web-address :type url)))
```

The user creates an instance of `professor` using the following steps:

1. Selects the `professor` node in the graphical display area and clicks on the icon. The form shown in figure 16 appears. Each slot in an instance form is represented by a row containing two or four columns. The first column contains a button with the slot name. Pressing the button causes all examples of the use of the slot to be displayed in the fine-grained view window. Figure 17 shows the output in the fine-grained view window after the “works-at” button has been pressed. The second column contains a text window into which a value can be typed. If the slot is typed a third and fourth column are provided. The third column is a menu containing the types defined for the slot and the subclasses of the defined types. The fourth column is a menu containing the instances of the selected type.
2. Starts to fill in the `works-at` slot. Examples of how the `works-at` slot has been used (shown in figure 17) are obtained by pressing the “works-at” button.
3. Selects “higher-educational-organization” from the type menu (see figure 18).
4. Selects “New Instance” from the instances menu (see figure 19). This causes a new instance form for the class `higher-educational-organization` to be created (see figure 20).
5. Fill in the instance form for the `higher-educational-organization` class.
6. Fill in the remainder of the `professor` instance slots and hits the “OK” button.

Finally, the user ends the session by selecting the “Edit Mode” menu item under the “Edit” menu. WebOnto indicates that the `persons-and-organizations` ontology is now unlocked. The appearance of WebOnto at the end of the session is shown in figure 21.

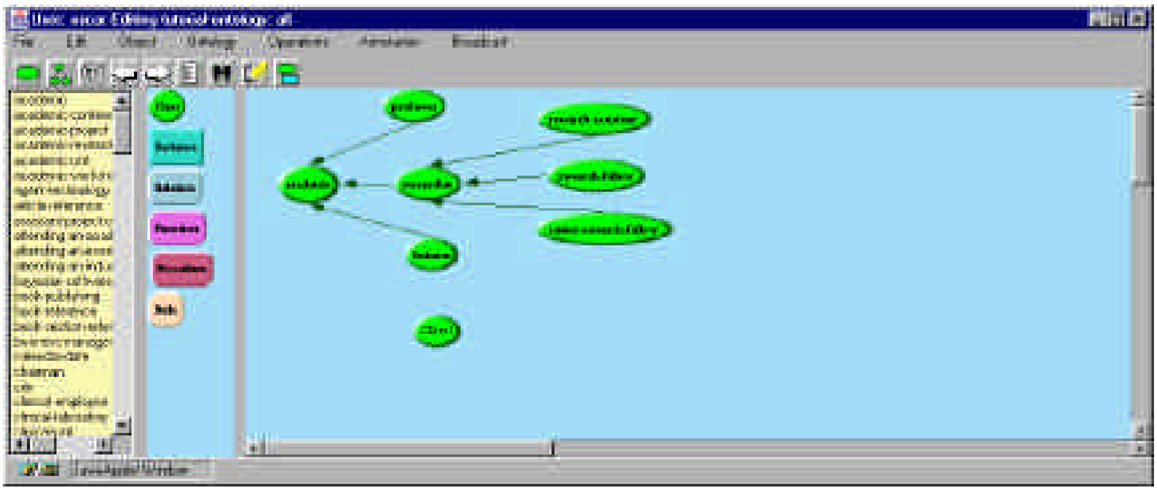
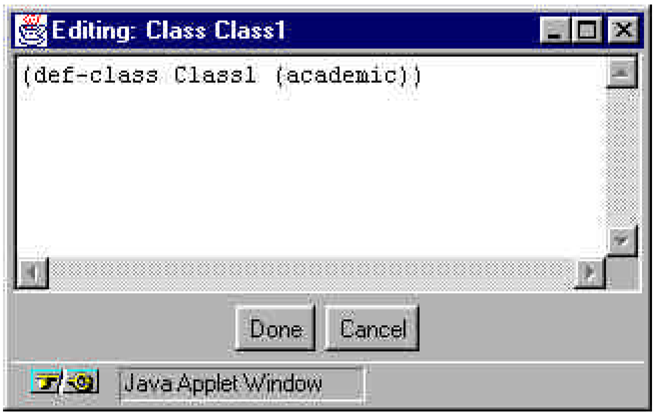


Figure 12. A screen snapshot showing a new class node `Class1` being dragged from the well.



```
(def-class Class1 (academic))
```

Figure 13. A screen snapshot showing the edit window created by selecting the `Class1` node shown in figure 9 and clicking on the ✎ icon.

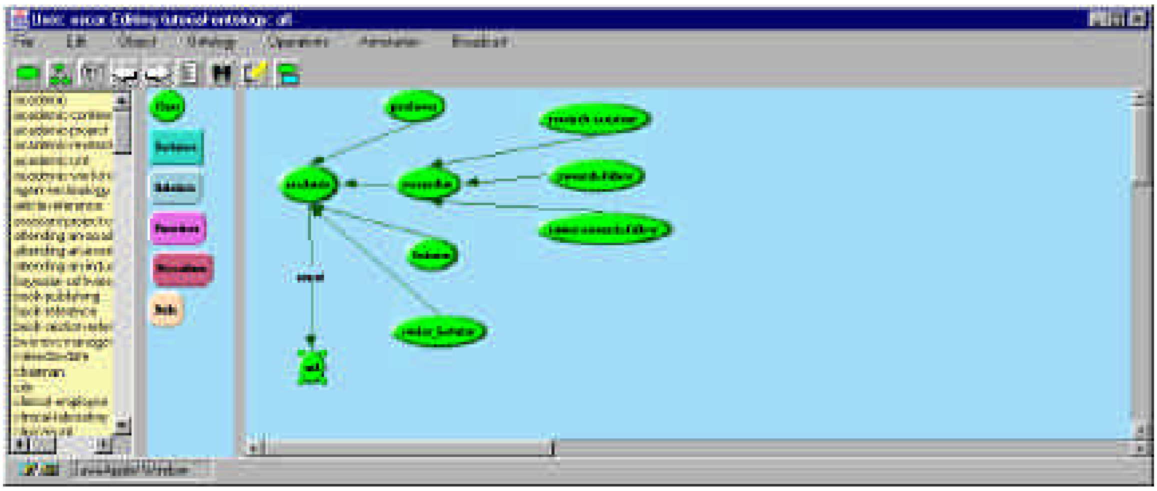


Figure 14. A screen snapshot just after the user has created a type link from the `academic` **class to the** `url` **class.**

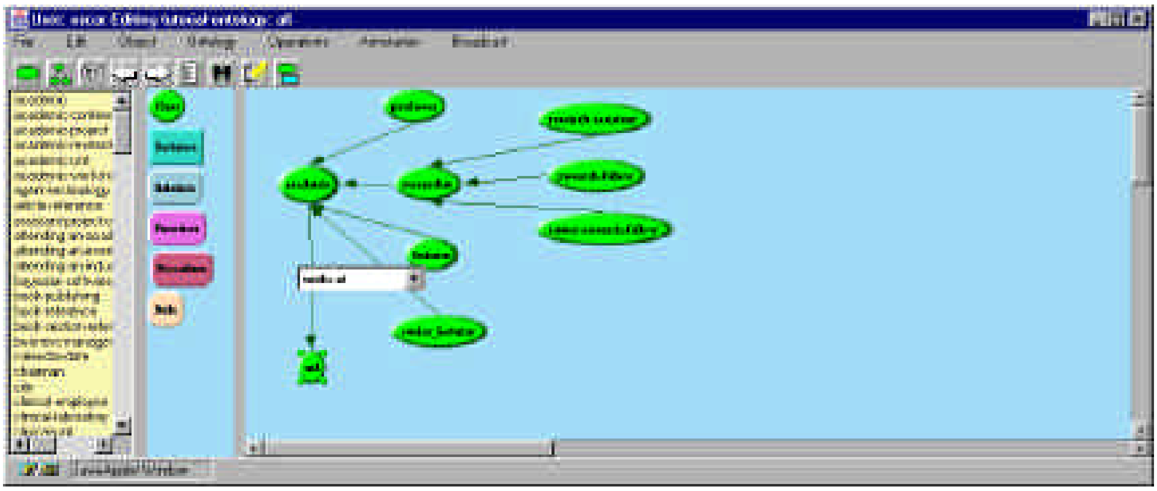
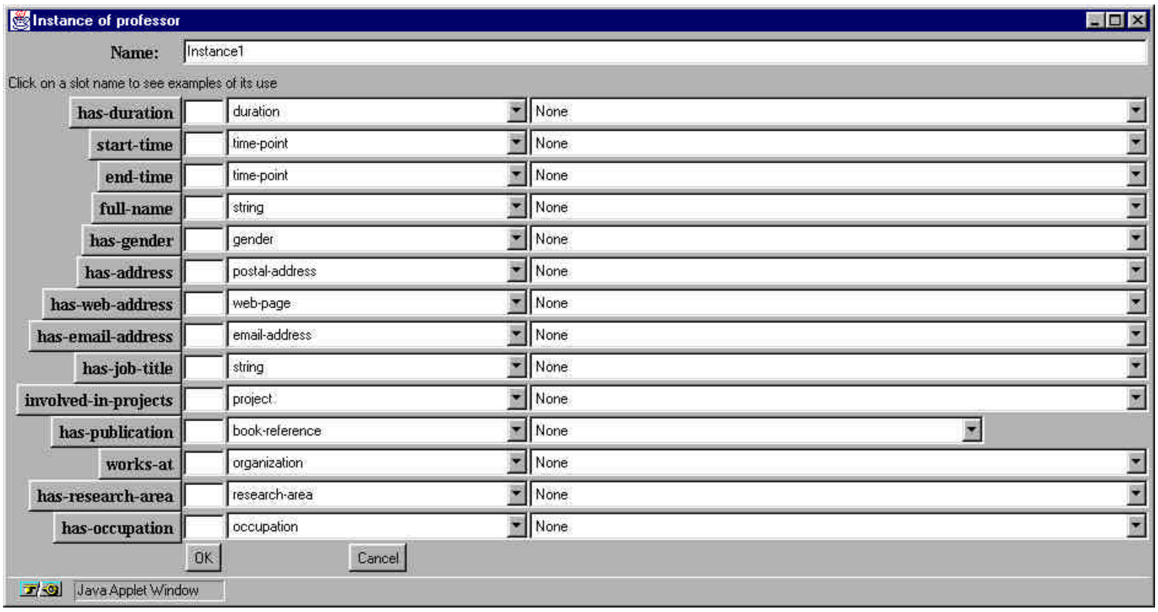


Figure 15. A screen snapshot just after the user has clicked on the “unset” label in the type link in figure 14 and before selecting “has-web-address”.

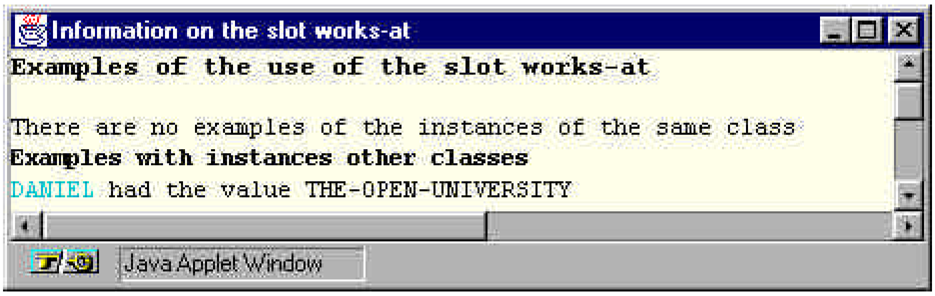


Name: Instance1

Click on a slot name to see examples of its use

has-duration	duration	None
start-time	time-point	None
end-time	time-point	None
full-name	string	None
has-gender	gender	None
has-address	postal-address	None
has-web-address	web-page	None
has-email-address	email-address	None
has-job-title	string	None
involved-in-projects	project	None
has-publication	book-reference	None
works-at	organization	None
has-research-area	research-area	None
has-occupation	occupation	None

Figure 16. A screen snapshot of the `professor` instance form automatically created by WebOnto.



```
Examples of the use of the slot works-at

There are no examples of the instances of the same class
Examples with instances other classes
DANIEL had the value THE-OPEN-UNIVERSITY
```

Figure 17. A screen snapshot of the fine-grained view window showing examples of the use of the slot `works-at`. This message was obtained by pressing on the "works-at" button in the instance form show in figure 16 above.

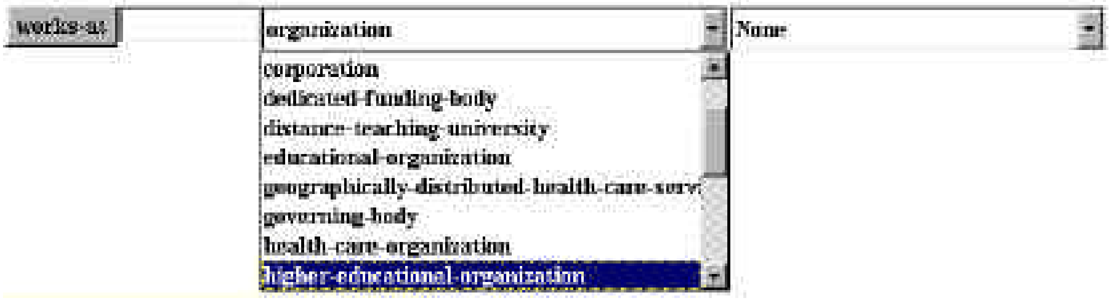
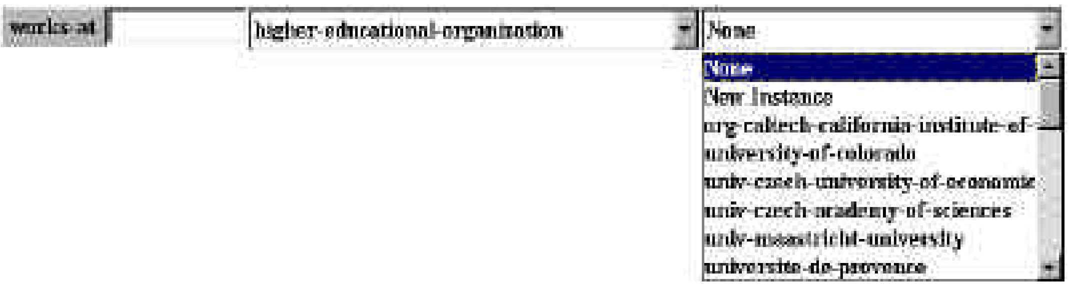


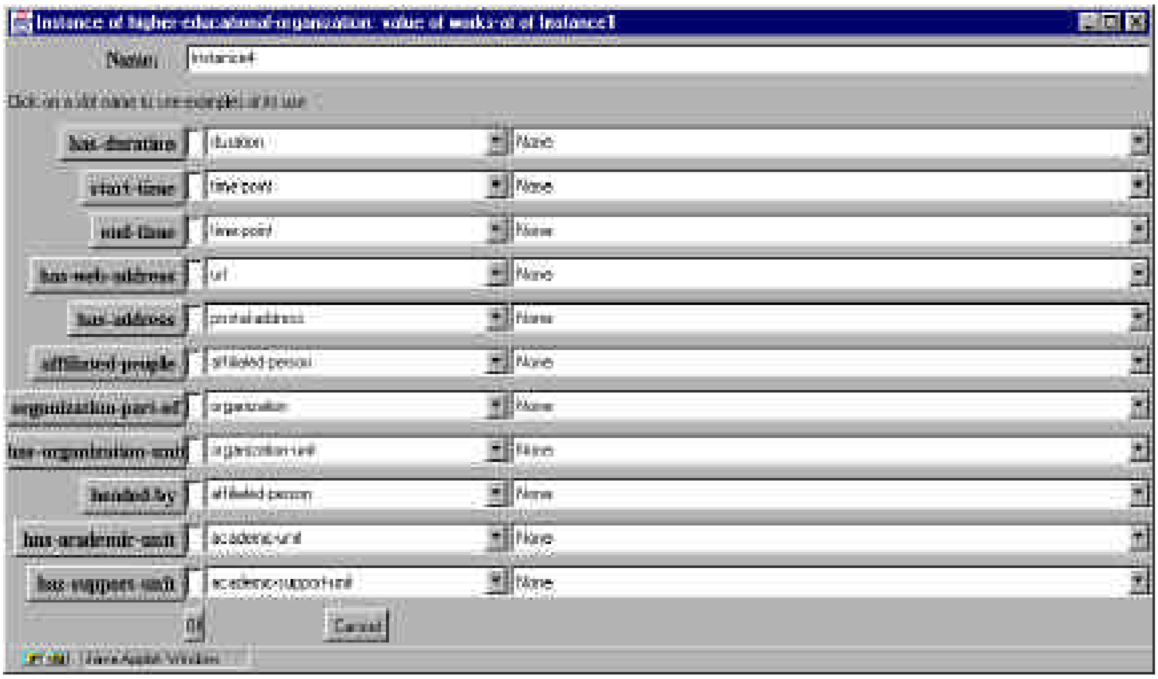
Figure 18. A screen snapshot of type menu for the `works-at` slot of the `professor` instance form.



works-at		higher-educational-organization

None
None
New Instance
org-caltech-california-institute-of-
university-of-colorado
univ-czech-university-of-economic
univ-czech-academy-of-sciences
univ-maastricht-university
universite-de-provence

Figure 19. A screen snapshot of the instance menu for the type `higher-educational-organization` **for the** `work-at` **slot of the** `professor` **instance form.**



Instance of higher-educational-organization: value of works-at of Instance1

Name: instance4

Click on a slot name to see examples of its use

has-duration	duration	None
start-time	time-point	None
end-time	time-point	None
has-web-address	url	None
has-address	postal-address	None
affiliated-people	affiliated-person	None
organization-part-of	organization	None
has-organization-unit	organization-unit	None
headed-by	affiliated-person	None
has-academic-unit	academic-unit	None
has-support-unit	academic-support-unit	None

Figure 20. A screen snapshot of the `higher-education-organization` **instance form for the slot** `works-at` **generated automatically by WebOnto.**

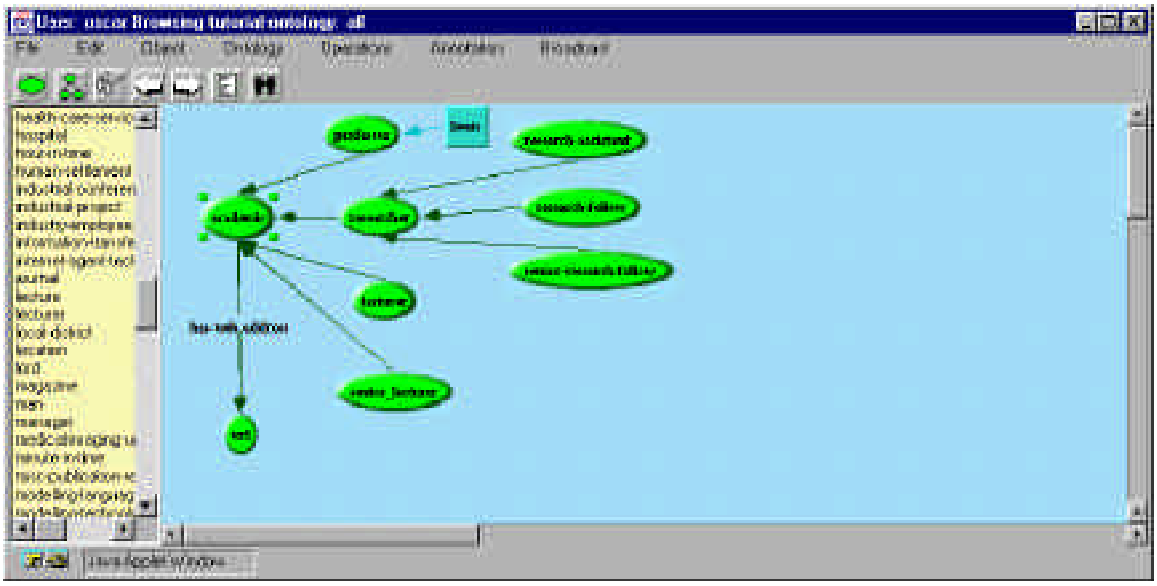


Figure 21. A screen snapshot showing WebOnto when the editing session has been terminated.

5.3 Summary of Mouse Gestures in the Graphical Area

Within WebOnto a number of operations can be carried out using mouse gestures in the Graphical Display window. The effect of a mouse gesture is dependent on WebOnto's current mode (browse or edit) and the gesture's start and end points. We summarise the possible operations in the following tables.

In Browse Mode

Gesture	*Start*	*End*	*Result*
Mouse Down	Node		Select the node
Control Mouse Down			Prompt for a text annotation
Mouse Drag	Node		Move a node
Mouse Drag	Not over a Node		Specify a select region

In Browse Mode with the "Trail" menu item under the "Annotation Menu" selected.

Gesture	*Start*	*End*	*Result*
Mouse Drag			Draw a freehand line

In Edit Mode

Gesture	*Start*	*End*	*Result*
Control Shift Mouse Down	Node		Create a text window to change the node name
Control Mouse Drag	Class Node1	Class Node2	Create a superclass link from node1 to node2
Control Mouse Drag	Instance Node1	Class Node2	Create an instance-of link from node1 to node2
Control Mouse Drag	Class Node1	Instance Node2	Create a default-value link from node1 to node2
Control (delayed Shift) Mouse Drag	Class Node1	Class Node2	Create a type-of link from node1 to node2

6 KNOWN BUGS

Currently there are three known bugs which can not be fixed trivially.

1. **Deselection of items -** Whenever an item is selected in either the item list, graphical area, or the fine-grained view window selected items in the other windows are deselected. Unfortunately, it is not possible to detect a selection using a single mouse click in the item list. You can select an item in the item list using a single mouse click but the event is not sent to WebOnto. Consequently, when an item is selected in the item list the currently selected items in the other two windows are not deselected.

2. Double clicking – Double clicking on an item in the item list or the fine-grained view window causes the item to be inspected. Double clicking is not always detected. This varies between hardware and operating system platforms.

3. No warnings before deleting a class - When deleting a class which has instances, there is no warning on the feature that every instance of the class is also deleted.

Chapter 2
Knowledge Modelling in OCML
Enrico Motta

The modelling language described in this chapter, OCML, was originally developed in the context of the VITAL project (Shadbolt et al., 1993), to provide operational modelling capabilities for the VITAL workbench (Domingue et al., 1993). Over the years the language has undergone a number of changes and improvements and in what follows I will present an overview of the current version of the language (v6.3) and compare it to alternative knowledge modelling languages. Moreover, I will also illustrate a subset of the application development interface supporting the construction of OCML models. This interface consists of a number of Lisp macros and functions which can be used for retrieving and modifying OCML constructs, for evaluating *functional* and *control terms*, and for querying an OCML model.

1 TYPES OF CONSTRUCTS IN OCML

OCML supports the specification of three types of constructs: *functional* and *control terms*, and *logical expressions*.

1.1 Functional terms

A functional term - in short, a term - specifies an object in the current domain of investigation. A functional term can be a *constant*, a *variable*, a *string*, a *function application* or can be constructed by means of a special *term constructor*. This can be one of the following: `if`, `cond`, `the`, `setofall`, `findall`, `quote` and `in-environment`[1] - see appendix 1 for a description of the semantics of these terms. Variables are represented as Lisp symbols beginning with a question mark - e.g., ?`x` is a variable. Strings are sequences of characters enclosed in double quotes, e.g. `"string"`. A function application is a term such as (*fun* {*fun-term*}*), where *fun* is the name of a function and *fun-term* a functional term. Functions are defined by means of the Lisp macro `def-function`, which is described in section 2.2.
Functional terms are evaluated by means of the OCML function interpreter.

1.2 Control terms

Modelling problem solving behaviour involves more than making statements and describing entities in the world. Control terms are needed to specify actions and describe the order in which these are executed. OCML supports the specification of sequential, iterative and conditional control structures by means of a number of control term

[1] In what follows I will use `this font` to refer to expressions in the OCML language.

constructors, such as `repeat`, `loop`, `do`, `if`, and `cond`[2]. A Lisp macro, `def-procedure`, makes it possible to label parametrized control terms - i.e. to define *procedures*. Control terms are evaluated by means of a control interpreter.

1.3 Logical expressions

OCML also provides the usual machinery for specifying logical expressions. The simplest kind of logical expression is a *relation expression*, which has the form (*rel* {*fun-term*}*), where *rel* is the name of a relation and *fun-term* is a functional term. More complex expressions can be constructed by using the logical operators - `and`, `or`, `not`, `=>`, `<=>` - and quantifiers - `forall` and `exists`. Operational semantics is provided for all operators and quantifiers. Relations are defined by means of the Lisp macro `def-relation`, which is described in section 2.1.

2 BASIC DOMAIN MODELLING IN OCML

In the previous section I have introduced the three types of constructs which are supported by OCML. In this section I will go down at a more fine-grained level of description and I will illustrate the various primitives which are provided in OCML to support the specification of logical expressions, functional and control terms. In particular, OCML provides mechanisms for defining *relations*, *functions*, *classes, instances*, *rules* and *procedures*.

2.1 OCML relations

Relations allow the OCML user to define labelled n-ary relationships between OCML entities. Relations are defined by means of a Lisp macro, `def-relation`, which takes as arguments the name of a relation, its *argument schema*, optional documentation and a number of *relation options*. An argument schema is a list (possibly empty) of variables. Relation options play two roles, one related to the formal semantics of a relation, the other to the operational nature of OCML. These roles are discussed in the next two sections.

2.1.1 Relation specification options

From a formal semantics point of view the purpose of a relation option is to help to characterize the extension of a relation. Table 2.1 shows the relation options which can be used to provide formal relation specifications and, for each option, informally describes its semantics. A formal semantics to these options can be given in terms of the homonymous Ontolingua constructs.

[2] The careful reader will have noticed that I have already mentioned `if` and `cond` when discussing functional terms. In fact, `if` and `cond` can be used to construct both functional and control terms. However this does not cause any problem. For instance, if an `if` construct is encountered when expecting a functional term then an error will be generated if control terms are used in the body of the `if`. The opposite however is not true: functional terms can always be used in place of control terms.

Relation Option	Role in Specification
`:iff-def`	Specifies both sufficient and necessary conditions for the relation to hold for a given set of arguments.
`:sufficient`	Specifies a sufficient condition for the relation to hold for a given set of arguments.
`:constraint`	Specifies an expression which follows from the definition of the relation and must be true for each instance of the relation.
`:def`	This is for compatibility with Ontolingua: it specifies a constraint which is also meant to provide a partial definition of a relation.
`:axiom-def`	A statement which mentions the relation to which it is associated. It provides a mechanism to associate theory axioms with specific relations.

Table 2.1. Relation specification options in OCML.

2.1.2 Operationally-relevant relation options

Relation options also play an operational role. Specifically, some relation options support *constraint checking* over relation instances while others provide *proof mechanisms* which can be used to find out whether or not a relation holds for some arguments. Table 2.2 lists the relation options which are meaningful from an operational point of view and informally describes their relevance to constraint checking and theorem proving.

As shown in the table, constraint checking is supported by the following keywords: `:constraint`, `:def` and `:iff-def`. While these have different model-theoretic semantics - see table 2.1 - from a constraint checking point of view they are equivalent. They all specify an expression which has to be satisfied by each known instance of the relevant relation.

The relation options `:iff-def`, `:sufficient`, `:prove-by` and `:lisp-fun` provide

Relation Option	Supports constraint checking	Provides proof mechanism
`:sufficient`	No	Yes
`:prove-by`	No	Yes
`:lisp-fun`	No	Yes
`:iff-def`	Yes	Yes
`:constraint`	Yes	No
`:def`	Yes	No

Table 2.2. Operationally-relevant relation options in OCML.

mechanisms for verifying whether or not a relation holds for some arguments. The first two - `:iff-def` and `:sufficient` - also play a specification role - see table 2.1. The others - `:prove-by` and `:lisp-fun` - only play an operational role.

Both `:iff-def` and `:sufficient` indicate logical expressions which can be used to prove whether some tuple is an instance of a relation. From a theorem proving point of view there is an important difference between them. Let's suppose we are trying to prove that a tuple, say T, satisfies a relation, say R. If a `:sufficient` condition is tried and failed, the OCML proof system will then search for alternative ways of proving that T satisfies R. If an `:iff-def` condition is tried and failed, then no alternative proof mechanism will be attempted.

The relation options `:prove-by` and `:lisp-fun` are meant to support rapid prototyping and early validation by providing efficient mechanisms for checking whether a tuple satisfies a relation. The difference between `:prove-by` and `:lisp-fun` has to do with the expressions which are used as values to the two options: `:prove-by` points to a logical expression, `:lisp-fun` to a non-logical one (specifically a Lisp expression).

The box below provides an example of how the various types of relation options can be used concurrently to specify a relation and to support constraint checking and efficient proofs. This example is taken from a task ontology for parametric design problems (Motta, 1999) – see http://chaucer.open.ac.uk:3000/webonto?ontology=parametric-design

The relation `has-value`, shown below, associates a *design parameter* to its value in a *design model*. The definition specifies that `?v` is the value of a parameter, `?p`, in a design model, `?dm`, if and only if the pair `(?p . ?v)` is an element of `?dm`. In addition, it also specifies the constraint that the value of a parameter has to be a member of its value range, if this has been specified. Finally, the definition includes a `:prove-by` option whose value is an expression which can be used for verifying whether the relation is satisfied for a triple, `(?p ?v ?dm)`. This expression provides an efficient proof method (by weakening the `:iff-def` statement which formally defines relation `has-value`) but does not contribute to the specification.

```
(def-relation HAS-VALUE (?p ?v ?dm)
  "Parameters have values w.r.t a particular design model"
  :iff-def (and (parameter ?p)
                (design-model ?dm)
                (element-of (?p . ?v) ?dm))
  :constraint (or (and (exists ?vr
                               (has-value-range ?p ?vr))
                       (element-of ?v ?vr))
                  (not (exists ?vr
                               (has-value-range ?p ?vr))))
  :prove-by (element-of (?p . ?v) ?dm))
```

2.1.3 A meta-option for non-operational specifications

As shown above, OCML provides a number of relation options, which play both a specification and an operational role. However, in some cases we might want to use a keyword only for specification and not operationally, for instance when we know that the value of the keyword in question is a non-operational expression. To cater for these situations, OCML provides a special meta-keyword, `:no-op`, which can be used to indicate that the enclosed relation option only plays a specification role. An example of its use is shown by the definition of relation `range`, which is shown below. In the example the keyword `:no-op` is used to indicate that the `:iff-def` specification of the relation is not operational - in particular it is not normally feasible to test the range of a function on all its possible inputs![3]

```
(def-relation RANGE (?f-r ?relation)
  "The range of a function or a binary relation is a relation which is
   true for any possible output of the function or second argument of
   the binary relation"
  :no-op (:iff-def (or
                    (and (function ?f-r)
                         (forall (?args ?result)
                                 (=> (= (apply ?f-r ?args) ?result)
                                     (holds ?relation ?result))))
                    (and (binary-relation ?f-r)
                         (forall (?x ?y)
                                 (=> (holds ?f-r ?x ?y))
                                     (holds ?relation ?y))))))
```

In the above definition it is worth highlighting the use of the special meta-relation `holds`. An expression such as (`holds` <r> <arg$_1$>....<arg$_n$>) is satisfied if and only if the expression (<r> <arg$_1$>....<arg$_n$>) is satisfied. Thus `holds` has variable arity: it can take one or more arguments. In particular the number of additional arguments in a `holds` statement reflects the arity of the relation passed as first argument. The relation `holds` has a ‘special status’ because it is the only relation with variable arity supported by OCML.[4]

2.1.4 OCML relations: summing up

The set of relation options discussed here aims to provide a flexible and versatile range of modelling constructs supporting various styles of modelling. While the emphasis is on operational modelling, OCML also supports formal specification. Moreover, it provides facilities for integrating a specification with efficient proof mechanisms.

[3] Ontolingua-aware readers may have induced from the definition of the relation `range` that, in contrast with Ontolingua, OCML does not considers functions as relations. This has to do with the operational nature of the language: functions are dealt with by the OCML interpreter, relations by the proof system.

[4] This constraint is only a limitation of the current implementation of the language: in principle there is no reason why variable-arity relations should not be supported. Moreover, in contrast with relations, OCML functions can have variable arity. To specify that a function can take an indefinite number of arguments OCML uses the same convention as Lisp: the symbol `&rest` is used before the last argument of a function argument schema.

2.2 OCML functions

A function defines a mapping between a list of input arguments and its output argument. Formally functions can be characterized as a special class of relations, as in KIF (Genesereth and Fikes, 1992). However, in operational terms there is a significant difference between a function and a relation: functions are applied to ground terms to generate function values; relation expressions can be asserted or queried. Thus, in accordance with the operational nature of OCML, functions are distinguished from relations.

Functions are defined by means of a Lisp macro, `def-function`. This takes as argument the name of a function, its argument list, an optional variable indicating the output argument (as in Ontolingua this is preceded by an arrow, ->), optional documentation and zero or more *function specification options*. These are `:def`, `:constraint`, `:body` and `:lisp-fun`.

The option `:constraint` provides a way to constrain the *domain* (i.e. the set of possible inputs) of a function. It specifies a logical expression which must be satisfied by the input arguments of the function. The `:def` option indicates a logical expression which 'defines' the function. This expression should be predicated over both (some) input arguments and the output variable. Operationally, the expression denoted by the `:constraint` option provides a mechanism for testing the feasibility of applying a function to a set of arguments. The expression denoted by `:def` provides a mechanism for verifying that the output produced by a function application is consistent with the formal definition of the function.[5]

Finally, the options `:body` and `:lisp-fun` provide effective mechanisms for computing the value of a function. The former specify a functional term which is evaluated in an environment in which the variables in the function schema are bound to the actual arguments. The latter makes it possible to evaluate an OCML function by means of a procedural attachment, expressed as a Lisp function. The arity of this Lisp function should be the same as that of the associated OCML function.

```
(def-function filter (?l ?rel) -> ?sub-l
  "Returns all the elements in ?l which satisfy ?rel"
  :def (and (unary-relation ?rel)
            (list ?l))
            (list ?sub-l)
            (=> (and (member ?x ?sub-l)
                     (holds ?rel ?x))
                (member ?x ?l)))
  :body (if (null ?l)
          ?l
          (if (holds ?rel (first ?l))
            (cons (first ?l)
                  (filter (rest ?l) ?rel))
            (filter (rest ?l) ?rel))))
```

[5] These constraint-checking mechanisms can be switched off, if they are not required.

The above definition shows an example of the use of `def-function`. The OCML function `filter` takes as arguments a list, `?l`, and a unary relation, `?rel`, and returns the elements of `?l` which satisfy `?rel`. As illustrated by the definition, the `:def` option provides a declarative way of specifying a function; the option `:body` an effective way of computing its value, for a given set of input arguments.

2.3 OCML classes

OCML also supports the specification of classes and instances and the inheritance of slots and values through *isa hierarchies*.

Classes are defined by means of a Lisp macro, `def-class`, which takes as arguments the name of the class, a list (possibly empty) of superclasses, optional documentation, and a list of *slot specifications*, as illustrated by the definitions in the next box. These show a number of classes taken from the domain model for the Sisyphus-I office allocation problem (Linster, 1994). This problem consists of allocating members of the YQT laboratory to the appropriate offices, according to a number of constraints.

```
(def-class YQT-member ()
  ((has-project :type project)
   (smoker :type boolean :cardinality 1)
   (hacker :type boolean :cardinality 1)
   (works-with :type YQT-member)
   (belongs-to-group :type research-group :value yqt)))

(def-class researcher (YQT-member))

(def-class secretary (YQT-member))

(def-class manager (YQT-member))
```

OCML provides support for the usual slot specification machinery which is found in frame-based languages. Specifically, it provides the following slot options.

`:value`. A value which is inherited by all instances of a class.

`:default-value` . A value which is inherited by all instances of a class, unless overridden by other values.

`:type`. The value of this option should be a class, say C. This option specifies that all values of the associated slot should be instances of C. It is also possible to specify that a slot value must belong to one or more classes, say `C1`,...., `Cn`, by using the notation `(or C1 C2..Cn)`.

`:max-cardinality` . The maximum numbers of slot values allowed for a slot.

`:min-cardinality` . The minimum numbers of slot values required for a slot.

`:cardinality`. The numbers of slot values required for a slot. This option subsumes both `:min-cardinality` and `:max-cardinality`.

`:documentation`. The value of this option is a string providing documentation for a slot.

`:inheritance`. The inheritance mechanism used for dealing with default values. If `:merge` is used, then all default values inherited from different ancestors are collected. If `:supersede` is used, then default values inherited from more specific ancestors override those inherited from more generic ones - see appendix 1 for more details on the inheritance mechanism in OCML.

2.4 OCML instances

Instances are simply members of a class. An instance is defined by means of `def-instance`, which takes as arguments the name of the instance, the *parent* of the instance (i.e. the most specific class the instance belongs to), optional documentation and a number of slot-value pairs. An example of instance definition, taken from the Sisyphus-I domain model, is shown in the box below.

```
(def-instance harry_c researcher
  ((has-project babylon)
   (smoker no)
   (hacker yes)
   (works-with jurgen_l thomas_d)))
```

In particular the above definition shows that a slot can have multiple values. In this case `harry_c` works both with `jurgen_l` and `thomas_d`.

2.5 Object-oriented and relation-oriented approaches to modelling

When describing classes and instances I made use of standard object-oriented terminology and talked about slots having values and instances belonging to classes. This object-centred approach is in a sense orthogonal to the relation-centred one which I used when discussing relations and logical expressions. The former focuses on the entities populating a model and then associates properties to them; the latter centres on the type of relations which characterize a domain and then uses these to make statements about the world. These two approaches to modelling/representation have complementary strengths and weaknesses and for this reason they are often combined in knowledge representation and modelling languages, to provide *hybrid* formalisms (Fikes and Kehler, 1985; Yen et al., 1988).

In the context of a knowledge modelling language (rather than a knowledge representation one) the main advantage gained from combining multiple paradigms is one of flexibility. Both object-oriented and relation-oriented approaches provide conceptual frameworks which make it possible to impose a view over some domain. The choice between one or the other can be made for ideological or pragmatic reasons - e.g. whether the target delivery environment is a rule-based shell or an object-oriented programming environment. In the specific context of an operational modelling language, such as OCML, another benefit, which is gained by providing support for both object-oriented and relation-oriented modelling, is that these approaches are naturally associated with particular types of inferences. Object-orientation provides the structure for inheritance and automatic classification; relation-orientation is normally associated with constraint-based and rule-based reasoning.

While the integration of multiple paradigms provides the aforementioned benefits, when describing or interacting with a knowledge model, it is useful to abstract from the various modelling paradigms and inference mechanisms integrated in the model and characterize it at a uniform level of description. Specifically, in accordance with a view of knowledge as a *competence-like* notion (Newell, 1982), it is useful to decouple the level at which we describe *what* an agent knows from the level at which we describe *how* the agent organizes and infers knowledge. Such an approach is used - for instance - in the Cyc system (Lenat and Guha, 1990), which integrates multiple knowledge representation techniques (the *heuristic* level), but provides a uniform interface to the Cyc knowledge base (the *epistemological* level). A similar approach is also followed by Levesque (1984), which describes a logic-based query language which can be used to communicate with a knowledge base at a functional level, independently from the data and inference structures present in the knowledge base.

In particular, in the case of OCML, this generic idea of providing a uniform level of description to a hybrid formalism has been instantiated by providing a *Tell-Ask* interface (Levesque, 1984), which use logical expressions (i.e. a relation-oriented view) when modifying or querying an OCML model, independently of whether the query in question concerns a class, a slot, or a ‘ordinary’ relation. The key to this integrated view is the fact that classes and slots are themselves relations; classes are unary relations and slots are binary ones. In addition to supporting a generic Tell-Ask interface, this property makes it possible to provide ‘rich’ specifications of classes and slots. In particular, because these are relations, it is possible to characterize them by means of the relation options discussed in section 2.1.1. For instance, the definition below specifies the class of empty sets in terms of an `:iff-def` relation option.

```
(def-class EMPTY-SET (set) ?set
   :iff-def (not (exists ?x (element-of ?x ?set))))
```

2.6 The generic Tell-Ask interface

2.6.1 Tell: a generic assertion-making primitive

OCML provides a generic assertion-making primitive, `tell`, which provides a uniform mechanism for asserting *facts*[6], independently of whether these refer to slot-filling assertions, new class instances, or simply relation instances. For example we can use `tell` to add a new value to the list of projects carried out by `harry_c` as follows.

[6] Formally a *fact* (or *assertion*) is a *ground relation expression.* A relation expression is an expression such as (<r> <arg$_1$>....<arg$_n$>), where <r> is a relation and arg$_i$ is a functional term. A ground expression (or term) is an expression (or term) which does not contain variables.

```
? (tell (has-project harry_c mlt))

(HAS-PROJECT HARRY_C MLT)
```

Analogously we can add a new instance of class researcher simply by stating:

```
? (tell (researcher mickey_m))

(RESEARCHER MICKEY_M)
```

2.6.2 Ask: a generic query-posing primitive

The relation-centred view makes it possible to examine the contents of an OCML model simply by asking whether a logical statement is satisfied. The OCML proof system will then carry out the relevant inference and retrieval operations, depending on whether the relation being queried is a slot, a class, or an ‘ordinary’ relation. The process is however transparent to the user. For instance, we can find out about the projects in which `harry_c` is involved - after the assertion shown above these are now `babylon` and `mlt` - by using the Lisp macro `ask` to pose the query `(has-project harry_c ?c).` The resulting interaction with the OCML proof system is shown below.

```
? (ask (has-project harry_c ?c))

Solution: ((HAS-PROJECT HARRY_C BABYLON))

More solutions?  (y or n)  y

Solution: ((HAS-PROJECT HARRY_C MLT))

More solutions?  (y or n)  y

No more solutions
```

This uniform, relation-centred view over OCML models also provides a way to index inferences. For instance, when answering the above query, the OCML proof system will first retrieve and order all inference mechanisms applicable to a query of type `has-project` - e.g., these might include assertions of type `has-project`, relation options associated with `has-project` and the relevant *backward rules*[7] - and will then try these in sequence, to generate one or more solutions to the query (more details on the OCML proof system are given in appendix 1).

[7] See section 2.8.1 for a description of OCML backward rules.

2.7 OCML procedures

Procedures define actions or sequences of actions which cannot be characterized as functions between input and output arguments. For example, the procedure below defines the sequence of actions needed to set the value of a slot. These include a `unassert` statement, which removes any existing value from the slot, and a `tell` statement, which adds the new value. Both `tell` and `unassert` are procedures. The former takes a ground logical expression and adds it to the current model. The latter takes a relation expression and removes from the current model all assertions which match it. Note that in accordance with the uniform view of a knowledge model, slot changes are carried out by means of generic assertion and deletion operations (i.e. in terms of `tell` and `unassert`).

```
(def-procedure SET-SLOT-VALUE (?i ?s ?v)
  :constraint (and (instance-of ?i ?c)
                   (slot-of ?s ?c))
  :body (do
          (unassert (list-of ?s ?i ?any))
          (tell (list-of ?s ?i ?v))))
```

2.8 Rule-based reasoning in OCML

2.8.1 Backward rules

OCML also supports the specification of *backward* and *forward* rules. A backward rule consists of a number of *backward clauses*, each of which is defined according to the following syntax:

backward-clause *::= (*relation-expression *{*`if` *{*logical-expression*}*$^+$*})*[8]

Each backward clause specifies a different *goal-subgoal decomposition*. When carrying out a proof by means of a backward rule the OCML interpreter will try to prove the relevant goal by firing the clauses in the order in which these are listed in the rule definition. As in Prolog, depth-first search with chronological backtracking is used to control the proof process.

[8] I use the following notational conventions when describing the syntax of OCML constructs. Braces, { and }, are used to indicate that the enclosed item is optional. For instance, the notation {x} means that x may or may not be present but, if it is present, it can only appear once. The notation {x}$^+$ means that x can appear 1 or more times (i.e. it must appear at least once), while the notation {x}* indicates that x can appear 0 or more times. Square brackets within braces are used in situations in which a number of alternatives for a *non-terminal* item are possible, but each alternative can be used only once. For example, let's consider a non-terminal item, x, for which alternatives a and b are possible. In this context the notation {[x] y}* indicates that any number of xy sequences are possible, but a or b can only appear once (in practice this means that we can have between 0 and 2 sequences). Braces can also be nested. For instance, the notation {x {y}} means that both x and y are optional but y can only appear if x does. Finally, I use italics to denote non-terminal items and a vertical bar, |, to indicate mutually exclusive alternatives.

Both semantically and operationally a backward chaining rule is the same as a `:sufficient` relation option: they both provide an expression which is sufficient to verify that a tuple holds for a relation. Thus, one might wonder whether rules are needed at all. In practice the advantage of including backward rules in the language is that these provide a modular mechanism for refining existing (possibly generic) relation specifications, for instance in cases where application-specific knowledge is needed to complete the specification of a relation. To clarify this point let's consider an example taken from the KMi office allocation problem. This was a real-world problem that our institute faced when moving to a new building, back in 1997.

The relation `has-value-range` is defined in the parametric design task ontology to characterize the set of possible values which can be assigned to a design parameter. When building an application model the generic `has-value-range` specification is usually refined, so that the space of possible values that can be assigned to a particular parameter is precisely defined. A modular way to achieve this is to refine the definition of the relation by means of the appropriate backward chaining rules. The rules shown below fulfil this purpose for two of the classes of parameters present in the KMI office allocation domain: professors and secretaries. In this example the former can only go into a double room; the latter into a large room next to the entrance.

```
(def-rule has-value-range-1
  ((has-value-range-gen ?m ?l)
   if
   (professor (domain-reference ?m))
   (= ?l (setofall ?r (double-a-type-room ?r usable yes)))))


(def-rule has-value-range-2
  ((has-value-range-gen ?m ?l)
   if
   (secretary (domain-reference ?m))
   (= ?l (setofall ?r
                   (and (room ?r size ?n usable yes)
                        (> ?n 1)
                        (close-to ?r kmi-entrance))))))
```

2.8.2 Forward rules

OCML also allows the user to define forward rules. A forward rule comprises zero or more antecedents and one or more consequents. Antecedents are restricted to relation expressions, while any logical expression can be a consequent. When a forward rule is executed, OCML treats each consequent as a goal to be proven and attempts to prove them, until one fails. This mechanism makes it possible to integrate data-driven and goal-driven reasoning and to specify arbitrarily complex right hand sides.

A special operator, `exec`, is provided to allow OCML users to introduce control (and therefore functional) terms in the right hand side of a rule. In particular, two useful procedures are `tell`, to assert new facts, and `output`, to produce output. A simple example showing how to use these in a forward chaining rule is given below.

```
(def-rule foo
  (has-project ?x ?y)
  then
   (exec (tell (project-covered-by ?y ?x)))
   (exec (output "has project ~S ~S" ?x ?y)))
```

While forward rules can be useful in a number of situations when building application models (e.g. to define *watchers*, which are triggered whenever some situation arises in a knowledge base), they are not essential to the model building process. The reason for this is that knowledge-level modelling is mainly about constructing definitions, while forward-chaining rules are about behaviour. Thus they can be used in place of procedures to describe behaviour but they cannot replace constructs for relation or function specification.

3 FUNCTIONAL VIEW OF OCML

A functional view of a knowledge representation system focuses on the services the system provides to the user (Levesque, 1984; Brachman et al., 1985). Basically, there are three kinds of services provided by OCML: i) operations for extending/modifying a model; ii) interpreters for functional and control terms; and iii) a proof system for answering queries about a model. Extensive details on the model extension/modification facilities provided by OCML were given in earlier sections.

4 MAPPING

OCML is meant to support both the specification of library components and the development of partial or complete application models, according to the *Task-Method-Domain-Application* (TMDA) framework described in (Motta, 1999). The role of the TMDA framework is to support application development by reuse and to this purpose it distinguishes between domain-generic *tasks* and *problem solving methods* on one side and domain(-specific) models on the other. Applications are created by instantiating generic problem solving components (tasks and methods) in a particular domain.

Earlier versions of the language provided support for task and method specification by means of special-purpose modelling constructs. The current version does away with these task and method-specific constructs and only provides a basic set of domain modelling facilities; the extension to the language required to specify tasks and methods is then defined as a particular representation ontology (the *task-method* sub-ontology of the OCML base ontology). The advantage of this approach is that it separates the core set of logical primitives (the OCML kernel) from additional, framework-specific epistemological commitments. Thus, only special-purpose primitives for defining *mapping knowledge* are required, in addition to ‘standard’ domain modelling capabilities. This mapping knowledge support application modelling, by allowing the customization of generic components in a particular domain. In the next sections I will discuss the two kinds of mapping constructs supported by OCML: *relation mapping* and *instance mapping*.

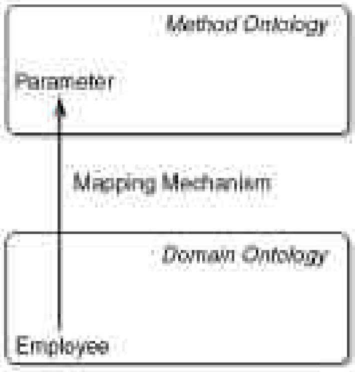


Figure 2.1. Mapping domain to method ontologies.

4.1 Instance mapping

Figure 2.1 illustrates a simple example in which a class of concepts at the task/method level (parameter) is mapped to a class of concepts at the domain level (employee). This is a very common situation when developing systems through reuse: a problem solving, domain-independent model imposes a particular view over a set of domain concepts (Fensel and Straatman, 1996).

A simple case is one in which a domain view is constructed by direct association of task level concepts to domain level concepts. For instance, a parametric design view over the Sisyphus-I domain can be imposed simply by creating parameter instances and associating them to YQT members. Thus, the set of parameters and the set of YQT members are associated but kept distinct. This solution is appealing for two reasons: it supports reuse of modular components and does not confuse the two different types of concepts; parameters and YQT members maintain different sets of properties and different semantics, thus avoiding a situation in which a design parameter has a wife and a YQT member has a value range.

Instance mapping is supported in OCML by means of the Lisp macro `def-upward-class-mapping`. This takes the names of two classes as arguments and associates each instance of the first class to a purpose-built instance of the second class. By default the relation `maps-to` is used to associate the task level instance to the domain level one.

In the Sisyphus-I application model we should then state:

```
(def-upward-class-mapping yqt-member yqt-parameter)
```

The above form iterates over each instance of class `yqt-member`, say `I`, creating a new instance of class `yqt-parameter`, say `Meta-I`, and associating this to `I`, i.e., asserting `(maps-to Meta-I I)`. Hence, if we now ask for the mapping between parameters and YQT members we get the following results.

```
? (ask (maps-to ?z ?x)t)

Solution: ((MAPS-TO YQT-PARAMETER-EVA_I EVA_I))

Solution: ((MAPS-TO YQT-PARAMETER-MONIKA_X MONIKA_X))

Solution: ((MAPS-TO YQT-PARAMETER-ULRIKE_U ULRIKE_U))

Solution: ((MAPS-TO YQT-PARAMETER-UWE_T UWE_T))

Solution: ((MAPS-TO YQT-PARAMETER-JOACHIM_I JOACHIM_I))

Solution: ((MAPS-TO YQT-PARAMETER-HANS_W HANS_W))

Solution: ((MAPS-TO YQT-PARAMETER-MICHAEL_T MICHAEL_T))

Solution: ((MAPS-TO YQT-PARAMETER-ANGY_W ANGY_W))

Solution: ((MAPS-TO YQT-PARAMETER-JURGEN_L JURGEN_L))

Solution: ((MAPS-TO YQT-PARAMETER-KATHARINA_N KATHARINA_N))

Solution: ((MAPS-TO YQT-PARAMETER-THOMAS_D THOMAS_D))

Solution: ((MAPS-TO YQT-PARAMETER-HARRY_C HARRY_C))

Solution: ((MAPS-TO YQT-PARAMETER-ANDY_L ANDY_L))

Solution: ((MAPS-TO YQT-PARAMETER-MARC_M MARC_M))

Solution: ((MAPS-TO YQT-PARAMETER-WERNER_L WERNER_L))
```

The advantage of this solution is that parameters and YQT members maintain their separate identities, as shown in the next box.

```
? (describe-instance 'YQT-PARAMETER-WERNER_L)

Instance YQT-PARAMETER-WERNER_L of class YQT-PARAMETER

HAS-VALUE-RANGE: (C5-123 C5-122 C5-121 C5-120 C5-119 C5-117 C5-116 C5-113 C5-
114 C5-115)

? (describe-instance 'WERNER_L)

Instance WERNER_L of class RESEARCHER

HAS-PROJECT: RESPECT
SMOKER: NO
HACKER: YES
WORKS-WITH: ANGY_W, MARC_M
GROUP: YQT
```

Formally, a mapping can be characterized as an association between an object, say `o`, and its meta-object, `m-o`, so that the entity denoted by the entity denoted by `m-o` is the same as the entity denoted by `o` (Genesereth and Nilsson, 1988). This notion can be formalized in OCML by means of the following definition.

```
 (def-relation maps-to (?x ?y)
  "This relation allows the user to specify an association between
   an object at the task layer and one at the domain layer.
   Formally ?y denotes the object denoted by the object denoted by ?x"
  :no-op (:iff-def (= ?y (denotation ?x))))
```

4.2 Relation mapping

Instance mapping works only in those cases in which imposing a view over a domain can be reduced to creating task-level ‘mirror images’ for a finite number of domain-level objects. A more general scenario is one in which there is some relation defined at the task/method level which needs to be reflected to the domain level in a dynamic fashion. A well known example is that in which domain concepts or statements are viewed as hypotheses at the problem solving level. This association is typically dynamic, given that hypotheses are considered as such only for a particular time-slice of the problem solving process. These situations can be modelled in OCML by means of *relation mappings*.
A relation mapping provides a mechanism to associate rules and procedures to a relation, say R, so that when a query of type R is posed, or assertions of type R are made at the task/method level, these events can be *reflected* to the domain level. The purpose of these reflection actions is to ensure that the consistency between domain and task/method levels is maintained.

An example of an *upward relation mapping* is illustrated by the definition below, which is taken from an application model developed for the Sisyphus-I problem. The mapping is an upward one, in the sense that it is used *to lift* (van Harmelen and Balder, 1992) the office allocation statements existing at the domain level to the problem solving level.

Specifically, the goal of this mapping is to associate the relation `current-design-model`, which is used by the parametric design problem solver to indicate the design model associated with the current design state, to the set of `in-room` assertions present in the current snapshot of the domain knowledge base. The relation `maps-to` is used to retrieve the parameter associated with each particular YQT member.

```
(def-relation-mapping current-design-model :up
   ((current-design-model ?dm)
    if
    (= ?dm (setofall (?p . ?v)
                    (and (in-room ?x ?v)
                         (maps-to ?p ?x))))))
```

An upward relation mapping ensures that when a task/method level relation is needed the relevant information is obtained from the domain level. Of course, problem solving is also about inferring knowledge and retracting previously held assertions. Hence, OCML also supports *downward relation mappings*. These divide into two categories, `:add` and `:remove`. The former specifies a procedure which is activated when a new relation instance is asserted. The latter specifies a procedure which is activated when a relation instance is removed. In the case of relation `current-design-model` relation mappings are needed to ensure that when the design model considered by the problem solver is modified, the relevant changes are reflected onto the domain model - see definition below.

```
(def-relation-mapping current-design-model (:down :add)
  (lambda (?x)
    (do
      (unassert (in-room ?any-m ?any-r))
      (loop for ?pair in ?x
            do
            (if (maps-to (first ?pair) ?z)
              (tell (in-room ?z (rest ?pair))))))))
```

Finally, the definition below shows the `:remove` downward mapping associated with relation `current-design-model`: it simply removes the domain level assertions associated with the design model which is passed as argument to the relation instance being retracted.

```
(def-relation-mapping current-design-model (:down :remove)
   (lambda (?x)
     (loop for ?pair in ?x
            do
            (if (maps-to (first ?pair) ?z)
              (unassert (in-room ?z (rest ?pair)))))))
```

5 ONTOLOGIES

OCML also provides support for defining ontologies. When an ontology is defined, say O, it is possible to specify which ontologies are included in O and as a result O will include all the definitions from its parent ontologies. When conflicts are detected (e.g., the same concept is defined in two different parent ontologies), a warning is issued. Primitives for loading and selecting ontologies are also provided.
By default all ontologies are built on top of the *OCML base ontology*. This comprises twelve sub-ontologies which include the basic definitions required to reason about basic data types (e.g. lists, numbers, sets and strings), the OCML system itself and the OCML frame representation. Specifically, the following sub-ontologies are provided:

Meta. This ontology defines the concepts required to describe the OCML language. It includes constructs such as ‘OCML expression’, ‘functional term’, ‘rule’, ‘relation’, ‘function’, ‘assertion’, etc. This ontology is particularly important to construct reasoning components which can verify OCML models.

Functions. Defines the concepts associated with functions - e.g., it includes relations such as `domain`, `range`, `unary-function`, `binary-function`, etc.

Relations. Defines the various notions associated with relations. These include the universe and the extension of a relation, the definition of reflexive and transitive relations, partial and total orders, etc.

Sets. This ontology defines the notions associated with sets - e.g., ‘empty set’, ‘union’, ‘intersection’, ‘set partition’, ‘set cardinality’, etc.

Numbers. Defines the various concepts and operations required for reasoning about numbers and for performing calculations.

Lists. Defines the concepts and operations associated with lists. It includes classes such as `list` and `atom`; functions such as `first`, `rest` and `append`; and relations such as `member`.

Strings. Specifies the concepts and operations associated with strings - e.g., `string`, `string-append`, etc.

Mapping. This ontology defines the concepts associated with the mapping mechanism described earlier. It includes only three definitions: relation `maps-to` and functions `meta-reference` and `domain-reference`. The former takes a domain-level instance and returns the associated task/method level instance. The latter performs the inverse function.

Frames. Defines the concepts associated with the frame-based representation used in OCML. It comprise (meta-) classes such as `class` and `instance`; functions such as `direct-instances` and `all-slot-values`; relations such as `has-one` and `has-at-most;` and procedures such as `append-slot-value`.

Inferences. The purpose of this ontology is to provide a repository for defining functions and relations supporting the specification of KADS-like inferences. So far only a few such inferences have been added to this ontology to support different types of selection and sorting.

Environment. This ontology provides a kind of 'environmental support' for the construction of OCML models. It includes special operators like `exec`, which makes it possible to invoke procedures from rules, and procedures such as `output`, which prints out a message.

Task-Method. This ontology provides the concepts required to model tasks and problem solving methods, i.e. to support the construction of task and problem solving models.

This set of ontologies provides a rich modelling platform from which to build other ontologies and/or problem solving models. It is natural to compare the OCML and Ontolingua base ontologies (Farquhar et al., 1996). There are two aspects which distinguish these two sets of ontologies: their nature and their scope.

The first difference is related to the operational nature of OCML. The Ontolingua base ontology is not concerned with operationality and therefore includes many non-operational definitions. The OCML base ontology is concerned with providing support for the construction of operational models. As a result it attempts to minimize the number of non-executable specifications. A typical approach is to weaken a non-operational definition to make it executable. For instance let's consider the function `universe`. In the Ontolingua base ontology the universe of a relation is defined as the set of all objects for which the relation is *true*. Of course this is not an operational definition. However, we can provide a weaker version of universe, called `known-universe`, which returns the set of all entities which are part of a tuple satisfying the relation in question. This function can be either defined separately from `universe` or attached to it to provide an operational definition.

The second difference concerns the scope of the two base ontologies. Both the Ontolingua and OCML base ontologies provide a rich set of definitions for domain modelling. In order to comply with the requirements imposed by the TMDA framework, the OCML base ontology provides support also for specifying tasks and problem solving methods.

6 COMPARISON WITH OTHER LANGUAGES

In the previous section I compared the base ontologies provided by OCML and Ontolingua and emphasized that the differences between them have mainly to do with the conceptual requirements imposed by the TMDA framework on OCML and with its operational nature. If we compare OCML and Ontolingua purely as modelling languages, their main difference has to do with the fact that while Ontolingua is concerned exclusively with ontologies - i.e. term specification - OCML aims to model behaviour as well. For this reason OCML also provides support for defining control terms. Apart from this aspect, the current version of OCML closely mirrors many of the constructs in Ontolingua. Thus, while OCML extends Ontolingua in various respects, it is possible (although reductive) to view OCML as an environment for prototyping

Ontolingua models, thus moving away from the batch-oriented, translation-based operationalization model suggested for Ontolingua (Gruber, 1993).

The design philosophy underlying the KARL language (Fensel, 1995) has many points in common with OCML. In particular both KARL and OCML are operational modelling languages and are therefore suitable for rapid prototyping of knowledge level models. Moreover, both KARL and OCML are part of comprehensive knowledge engineering frameworks providing methodological support for KBS development. However, there are also differences. KARL is an executable, formal specification language where the emphasis is on formalization: its main strength is the provision of a formal semantics for its modelling constructs. The design philosophy of OCML has more to do with pragmatic considerations: its main goal is to provide a flexible modelling environment able to support different approaches to modelling: rapid prototyping and structured development, executable and non-executable constructs, formal and informal specifications. The two languages also differ in terms of the modelling frameworks they employ: KARL is based on the KADS four-layer framework, while OCML (more precisely the task-method ontology) is based on the TMDA framework. An important difference is also that while the primitives in the KARL language closely reflect the KADS approach, the OCML kernel is approach-neutral. Its commitment to the TMDA framework is defined by means of the appropriate ontology. This approach has the advantage of flexibility: different modelling frameworks can then be supported through the specification of the relevant ontologies.

A more recent version of KARL, which is called *New-KARL* (Angele et al., 1996), does away with the strong KADS-oriented approach used by KARL and focuses instead on the specification of task-method structures and ontology mappings. Thus New-KARL subscribes to a modelling framework which has much more in common with the OCML task ontology than the one underlying KARL. However, in contrast with OCML, New-KARL also ‘hardwires’ these task and method-centred primitives in the language itself, rather than in a particular ontology.

Other formal specification languages exist for KADS models - see (Fensel and van Harmelen, 1994) for an overview. While the formal details of these languages of course vary, similar conclusions to those drawn above can be reached when comparing them to OCML: these languages tend to emphasize formal aspects and are based on a KADS approach. OCML emphasizes operationality and flexibility and does not presuppose (although it can support) a KADS approach. Indeed, the original raison d’être for OCML was to provide an operational alternative to a formal specification language, $K_{BS}SF$ (Jonker and Spee, 1992), in the context of the VITAL workbench (Domingue et al., 1993).

Among the informal notations available for knowledge modelling the most notable is the CML language (Schreiber et al., 1994), which supports ontological specifications and the construction of Common KADS models. CML supports the definition of various constructs, including concepts, attributes, tasks, methods, relations, structures and expressions. Obviously, the main difference between CML and OCML is that the former is only meant to be an informal notation while the latter is a fully operational language. Another important difference is that CML is committed to supporting the Common

KADS framework, while the kernel of the OCML language is framework-independent. In addition to the KADS-related commitments CML also embodies other modelling commitments: it provides primitives for representing structures and part-of relations. This approach has both advantages and disadvantages. On the plus side it extends the range of modelling primitives provided by the language and supports notions which occur frequently in conceptual modelling. On the other hand there are two possible problems with this approach. The first one has to do with embedding ontological commitments in a conceptual modelling language. In particular different approaches to modelling structure and aggregation can be found in the literature - e.g. compare the analysis by Martin and Odell (1995) with that by Lenat and Guha (1990). Embedding one particular approach in the kernel of a conceptual modelling language prevents users from extending the language according to an alternative approach. The second problem is caused by the provision of different levels of description - i.e. logical, epistemological and conceptual (Brachman, 1979) - within the same formalism. Much work in knowledge representation over the past twenty years has focused on identifying the different levels at which knowledge representation languages can be specified (Brachman, 1979; Guarino 1994). In particular the paper by Brachman clearly illustrates that much of the confusion surrounding the field of knowledge representation in the seventies was caused by the fact that researchers were comparing formalisms which were situated at different levels - e.g. logical and conceptual. Thus, it seems to me that including ontological primitives (e.g. structures) in a language characterized at the logical level is a potential source of confusion. This is especially the case with informal languages, given that eventual ambiguities are not explained by the underlying formal theory.

The LOOM language (MacGregor, 1991) is strictly speaking a knowledge representation rather than a knowledge modelling language (i.e., it also includes symbol-level representation constructs). However, its formal kernel - i.e., what are called the *terminological* and *assertional* components - provides purely logical and epistemological primitives and therefore it can be used for knowledge modelling and ontology specification. The main feature of LOOM is its powerful classification mechanism which integrates a sophisticated concept definition language with rule-based reasoning. This approach allows a wider range of inferences to be drawn than those available from ‘traditional’ frames+rules systems such as KEE. The existence of a powerful classifier is an important advantage that LOOM maintains over OCML. On the other hand, viewed purely as a knowledge modelling language, LOOM exhibits a number of limitations.[9] When building knowledge models it may be necessary to make statements about the model being developed, which do not have a direct inferential purpose. For this reason OCML allows the inclusion of non-operational statements and supports the specification of axioms about the current model. In contrast, LOOM only provides operational

[9] The following points should not be construed as criticisms of the LOOM language. This is primarily a knowledge representation language and should be of course judged with respect to knowledge representation criteria. However, given the high-level of support provided by the language, its ‘organic relationship’ with the Ontolingua effort and its sound theoretical basis, it makes sense to consider it as a plausible candidate for knowledge modelling. Indeed LOOM has been used for ontological work (Swartout et al., 1996).

constructs. Another possible problem related to LOOM is that while it supports different knowledge representation paradigms (frames, rules, message-passing) it integrates the various constructs according to a classification-centred viewpoint. While this approach obtains nice results in terms of inferential capabilities, it nevertheless can be a constraint for the knowledge analyst, who is forced to frame the current problem within a classification-centred framework. In contrast with this approach OCML provides a number of alternative modelling constructs, e.g. rules, functions, classes and procedures which, while integrated, are themselves ‘primitive modelling components’ and can be used within different modelling approaches.

The notion of mapping presented in this chapter plays a role similar to that of *mediators* in heterogeneous information systems (Wiederhold and Genesereth, 1997), *connectors* in software architectures (Shaw and Garlan, 1996), and *adapters* in design patterns (Gamma et al., 1995). The idea underlying all of these approaches is essentially the same: some kind of ‘external kit’ is required in order to allow the interaction of reusable components and their configuration for different computational scenarios. The externalization of this adaptation process has the advantage that the original components remain unchanged, while they become usable in the new situation.

7 SYNOPSIS

The role of the OCML language is to provide operational knowledge modelling facilities. To this end it includes interpreters for functional and control terms, as well as a proof system which integrates inheritance with backward chaining, function evaluation and procedural attachments. While the emphasis here is on operationality, OCML aims to support different styles of knowledge modelling. Therefore OCML provides for first-order logic definitions and allows the user to explicitly distinguish non-operational from operational definitions. Moreover, it supports an extensive set of Ontolingua constructs and therefore can be used as an interpreter for Ontolingua definitions.

References

Angele, J., Decker, S., Perkuhn, R. and Studer, R. (1996). Modeling Problem Solving Methods in New KARL. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Alberta, Canada.

Brachman, R. J. (1979). On the Epistemological Status of Semantic Networks. In N. V. Findler (Editor), *Associative Networks: Representation and Use of Knowledge by Computers*. Academic Press, New York, pp. 3-50.

Brachman, R. J., Fikes, R. E. and Levesque, H. J. (1985). KRYPTON: A Functional Approach to Knowledge Representation. In R. J. Brachman and H. J. Levesque (Editors). *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, CA.

Domingue, J., Motta, E. and Watt, S. (1993) The Emerging Vital Workbench. In Ed. Aussenac, N., Boy, G., Gaines, B., Linster, M., Ganascia, J.-G. and Kodratoff, Y. Knowledge Acquisition for Knowledge-Based Systems 7th European Workshop, EKAW'93 Toulouse and Caylus, France, September, pp. 320-339, Springer-Verlag.

Domingue J. (1998) Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web. Proceedings of the 11th Banff Knowledge Acquisition Workshop, Banff, Alberta, Canada, April 18-23, 1998.

Farquhar, A., Fikes, R., and Rice, J. (1996). The Ontolingua Server: A Tool for Collaborative Ontology Construction. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Alberta, Canada.

Fensel, D. (1995). *The Knowledge Acquisition and Representation Language KARL*. Kluwer, Dordrecht.

Fensel, D. and van Harmelen, F. (1994). A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise. *The Knowledge Engineering Review, 9(2).*

Fensel, D. and Straatman, R. (1996). Problem solving methods: Making Assumptions for Efficiency Reasons. In N. Shadbolt, K. O'Hara, and Schreiber, G. (Editors). *Advances in Knowledge Acquisition - EKAW '96*. Lecture Notes in Artificial Intelligence, 1076. Springer-Verlag, Heidelberg.

Fikes, R. E. and Kehler, T. (1985). The Role of Frame-Based Representation in Reasoning. *Communications of the ACM, 28(9)*, September 1985.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns.* Addison-Wesley.

Genesereth, M. R. and Fikes, R. E. (1992). Knowledge Interchange Format, Version 3.0. *Technical Report Logic-92-1*, Computer Science Department, Stanford University.

Genesereth, M. R. and Nilsson, N. J. (1988). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, CA.

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition, 5(2)*.

Guarino, N. (1994). The Ontological Level. In R. Casati, B. Smith and G. White (Editors), *Philosophy and the Cognitive Science*. Holder-Pichler-Tempsky, Vienna, Austria.

van Harmelen, F. and Balder, J. R., (1992). (ML)^2: A Formal Language for KADS Models of Expertiese. *Knowledge Acquisition, 4(1)*, pp. 127-161.

Jonker, W. and Spee, J. W. (1992). Yet Another Formalization of KADS Conceptual Models. In Th. Wetter, K.-D. Althoff, J. Boose, B.R. Gaines, M. Linster and F. Schmalhofer (Editors) *Current Developments in Knowledge Acquisition - EKAW '92*, pp. 112-32. LNAI 599, Springer-Verlag, Berlin.

Lenat, D.B. and Guha, R.V. (1990). *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*. Addison-Wesley, Reading, MA.

Levesque, H. J. (1984). Foundations of a functional approach to knowledge representation. *Artificial Intelligence 23(2)*, pp. 155-212.

Linster, M. (1994). Problem Statement for Sisyphus: Models of Problem Solving. *International Journal of Human-Computer Studies 40(2)*, pp. 187-192.

MacGregor, R. (1991). Using a Description Classifier to Enhance Deductive Inference. *Proceedings of the 7th IEEE Conference on AI Applications*. Miami, Florida, February 1991.

Martin, J. and Odell, J. J. (1995). *Object-Oriented Methods: A Foundation.* Prentice-Hall, Englewood Cliffs, New Jersey.

Motta E. *Reusable Components for Knowledge Models: Principles and Case Studies in Parametric Design*. IOS Press, 1999.

Newell A. (1982). The knowledge level. *Artificial Intelligence, 18(1)*, pp. 87-127.

Riva, A. and Ramoni, M. (1996) LispWeb: a Specialised HTTP Server for Distributed AI Applications. Computer Networks and ISDN Systems, 28,7-11 (1996), 953-961.

Schreiber, A.Th., Wielinga B. J., Akkermans, H., van de Velde, W., and Anjewierden, A. (1994). CML: The CommonKADS Conceptual Modelling Language. In L. Steels, A. T. Schreiber, and W. van de Velde (Editors), *A Future for Knowledge Acquisition, Proceedings of the 8th European Knowledge Acquisition Workshop*. Springer Verlag, LNAI 867, pp. 283-300.

Shadbolt, N., Motta, E., and Rouge, A. (1993) Constructing Knowledge-Based Systems. IEEE Software, 10, 6, pp. 34-39, November 1993.

Shaw, M. and Garlan, D. (1996). *Software Architectures. Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

Wiederhold, G. and Genesereth, M. (1997). The Conceptual Basis for Mediation Services, *IEEE Expert, September/October Issue*, pp. 38-47.

Yen, J., Neches, R. and MacGregor, R. (1988). Classification-based Programming: A Deep Integration of Frames and Rules. *Technical Report ISI/RR-88-213*. USC/Information Science Institute, March 1988.

Appendix 1
Additional Details on OCML

1 FUNCTIONAL TERM CONSTRUCTORS

A BNF specification of each term constructor is provided as well as an informal description of its operational semantics.

`Setofall`

`setofall` finds all solutions (i.e. *environments*) to *basic-log-expression* and then returns the list obtained by instantiating *template* in all the returned environments, ensuring that the list contains no duplicates. If no solutions are found then the empty list (i.e. `nil`) is returned.

setofall-term ::= `setofall` template basic-log-expression

template ::= `nil` | (term . term)

term ::= constant | variable | string | (fun {term}*) | findall-term | the-term | in-env-term | quote-term | if-term | cond-term

fun ::= *the name of a function or a term constructor*

constant ::= *A symbol whose first character is not '?'*

variable ::= *A symbol whose first character is '?'*

string ::= *A lisp string, e.g.* "`string`".

log-expression ::= quant-log-expression | basic-log-expression

quant-log-expression ::= (`forall` schema-or-var log-expression)| (`exists` schema-or-var log-expression)

basic-log-expression ::= (`and` {log-expression}+) | (`or` {log-expression}+) | (`=>` log-expression log-expression) | (`<=>` log-expression log-expression) (`not` log-expression) | rel-expression schema-or-var ::= schema | variable

schema ::= *(*variable . schema*)* | nil

rel-expression ::= *(*rel *{*term*}***)*

rel ::= *a symbol naming a relation*

Findall

findall is the same as setofall except that it does not remove duplicate solutions.

findall-term ::= findall template basic-log-expression

The

the finds one solution (i.e. environment) to *basic-log-expression* and then returns the list obtained by instantiating *template* in the returned environment. If no solutions are found then the constant :nothing is returned.

the-term ::= the template basic-log-expression

In-environment

The primitive in-environment takes a list, possibly empty, of pairs ((var_1 . $term_1$)...) and a *body*, and returns the result of evaluating this in an environment in which each var_i is bound to $term_i$.

in-env-term ::= in-environment pairs body

pairs ::= nil | *(*pair . pairs*)*

pair ::= *(*variable . term*)*

body ::= term

Quote

The value of an expression such as (quote *term*) is *term*.

quote-term ::= 'term | *(*quote term*)*

If

The first action which is carried out when evaluating an *if-term* is to check whether *log-expression* is satisfied. If this is the case, then *then-term* is evaluated in the environment which satisfies *log-expression*. If *log-expression* cannot be satisfied in the current model, then there are two possibilities. If *else-term* is specified, then this is evaluated, and the value obtained is returned as the value of the *if-term*. If *else-term* is not present and *log-expression* cannot be proved, then the constant :nothing is returned.

if-term ::= *(*`if` log-expression then-term *{*else-term*})*

then-term ::= term

else-term ::= term

`Cond`

The interpreter iterates through each clause of a *cond-term*, until it finds one whose *log-expression* is satisfied. If none is found, then `:nothing` is returned. Otherwise, let's assume *cond-clause*$_i$. is the first clause whose *log-expression* is satisfied. In this case the value of the *cond-term* is obtained by evaluating the term associated with *cond-clause*$_i$.

cond-term ::= *(*cond *{*cond-clause*}*$^+$*)*

cond-clause ::= *(*log-expression term*)*

2 CONTROL TERM CONSTRUCTORS

A BNF specification of each control term constructor is provided as well as an informal description of its operational semantics.

`In-environment`

The primitive `in-environment` takes a list, possibly empty, of pairs ((*var*$_1$. *term*$_1$)...) and a *control-body*, and returns the result of evaluating this in an environment in which each *var*$_i$ is bound to *term*$_i$.

in-env-control-term ::= `in-environment` pairs control-body

pairs ::= `nil` | *(*pair . pairs*)*

pair ::= *(*variable . term*)*

control-body ::= control-term

control-term ::= term | in-env-control-term | if-control-term | cond-control-term | do-control-term | loop-control-term | repeat-control-term | return-control-term

if-control-term ::= *(*`if` log-expression then-control-term *{*else-control-term*})*

then-control-term ::= control-term

else-control-term ::= control-term

cond-control-term ::= (cond {cond-control-clause}$^+$)

cond-control-clause ::= (log-expression control-term)

loop-control-term ::= (`loop` `for` variable `in` term `do` {control-term}$^+$)

do-control-term ::= (`do-actions` {control-term}$^+$)

repeat-control-term ::= (`repeat-actions` {end-test} {control-term}$^+$) | (`repeat-actions` {control-term}$^+$ {end-test})

end-test ::= `while` test | `until` test

test ::= log-expression

return-control-term ::= (`return` term)

`If`

The first action which is carried out when evaluating an *if-control-term* is to check whether *log-expression* is satisfied. If this is the case, then *then-control-term* is evaluated in the environment which satisfies *log-expression*. If *log-expression* cannot be satisfied in the current model, then there are two possibilities. If *else-control-term* is specified, then this is evaluated, and the value obtained is returned as the value of the *if-control-term*. If *else-control-term* is not present and *log-expression* cannot be proved, then the constant `:nothing` is returned.

if-control-term ::= (`if` log-expression then-control-term {else-control-term})

`Cond`

The interpreter iterates through each clause of a *cond-control-term*, until it finds one whose *log-expression* is satisfied. If none is found, then `:nothing` is returned. Otherwise, let’s assume *cond-clause*$_i$. is the first clause whose *log-expression* is satisfied. In this case the value of the *cond-control-term* is obtained by evaluating the control-term associated with *cond-clause*$_i$.

cond-control-term ::= (cond {cond-control-clause}$^+$)

`Loop`

The control construct `loop` provides a simple mechanism for iterating over lists. It first evaluates a *term*, which should return a list, say *L*. Then it iterates over each element of *L*, say *I*, and evaluates *control-term* in an environment in which *variable* is bound to *I*.

loop-control-term ::= (`loop` `for` variable `in` term `do` {control-term}$^+$)

`Do`

The control construct `do` is a simple sequencing primitive. The control terms in its body are evaluated sequentially, once only.

do-control-term ::= (`do-actions` {control-term}$^{+)}$

`Repeat`

The control term constructor `repeat` repeats the control term(s) specified in its body until the end test is satisfied, if the test has the form '`until` test'. Otherwise, if the test has the form '`while` *test*', then `repeat-actions` stops as soon as the test fails. If the end test is specified after the control terms, then the control terms are carried out at least once - i.e. the end test is verified at the end of each cycle. If the end test is specified before the control terms, then the test is verified at the beginning of each cycle. If no test is provided, then all control expression in the body of a `repeat-actions` are repeated *ad infinitum*.

repeat-control-term ::= (`repeat-actions` {end-test} {control-term}$^{+}$) |
(`repeat-actions` {control-term}$^{+}$ {end-test})

`Return`

This is a simple way of exiting from the body of a `loop` or `repeat` construct. When a control term such as (`return` *term*) is encountered, the most specific loop or repeat construct in the current execution stack is exited and the value obtained from evaluating *term* is returned.

return-control-term ::= (`return` term)

3 INHERITANCE AND DEFAULT VALUES

Generally speaking default values are values which apply unless other alternatives can be used. In the OCML language the notion of default value is operationalized as follows.
Instances inherit values and default values from their superclasses down the inheritance hierarchy specified by `instance-of` and `subclass-of` links. For a given slot, say `s`, of a sample instance, say `I`, the following scenarios can arise:

1. `I` has not inherited any default value. In this case the value of s in I is given by all the values I has inherited from its superclasses, plus any value locally specified for slot s of I.
2. I has inherited some default values as well as non-default ones. In this case the default values are ignored and rule (i) is applied. We say that the default values are overridden by the non-default ones.
3. I has inherited only default values and local values have been specified. As in the previous case, the default values are ignored and only the local values are considered.
4. I has inherited only default values and no local values have been specified. In this case there are two possibilities. If the :inheritance facet has not been specified, or it has been specified and it is :merge, all default values apply. If the :inheritance facet has been specified and it is :supersede, then the value of s in I is obtained by (i) ranking the ancestors of I according to the class precedence order of the parent of I, and (ii)

retrieving the default value of the first class in the class precedence order which specifies a (default) value for s. The details of the algorithm used to compute the class precedence order are given at pp. 782-786 of the Common Lisp specification (Steele, 1992). This algorithm produces a total order (if this exists) based on two ordering principles: (i) a class, say C, precedes all its direct superclasses, and (ii) a direct superclass of `c` precedes the direct superclasses of `c` specified to its right in the list of direct superclasses of `c`.

4 INTERPRETERS AND PROOF SYSTEM

4.1 The OCML interpreter for functional terms

The OCML interpreter is implemented by means of a Lisp macro, `ocml-eval`. This evaluates a functional term, *term*, in an environment, *env*, according to the following rules.

1. If *term* is a variable, then the binding of *term* in *env* is returned.
2. If *term* is a string or a constant, then *term* is returned.
3. If *term* has the format (*pfun* $term_0$, ..., $term_n$), with n ≥ 0, where *pfun* is a *primitive term constructor*, then *term* is evaluated in *env*, according to criteria which depend on *pfun*.
4. If *term* has the format (*fun* $term_0$, ..., $term_n$), with n ≥ 0, where fun is the name of a function, and a Lisp body associated with *fun* exists, then `ocml-eval` returns the value obtained by applying the Lisp body to the values obtained by evaluating each $term_i$ in *env*.
5. If *term* has the format (*fun* $term_0$, ..., $term_n$), with n ≥ 0, where *fun* is the name of a function, and no Lisp body associated with *fun* exists, then `ocml-eval` returns the value obtained by applying the body of *fun* to the values obtained by evaluating each $term_i$ in *env*.
6. In all other cases `ocml-eval` signals an error.
7. The OCML interpreter for control terms
8. Control terms are interpreted in a manner analogous to functional terms. The control term interpreter is implemented by a Lisp macro, `ocml-control-eval`, which has the following behaviour.
9. If *term* is a functional term, then it is evaluated according to the rules given in section 4.1.
10. If *term* has the format (*proc* $term_0$, ..., $term_n$), with n ≥ 0, where *proc* is a *primitive control operator*, then *term* is evaluated in *env*, according to criteria which depend on *proc*.
11. If *term* has the format (*proc* $term_0$, ..., $term_n$), with n ≥ 0, where *proc* is the name of a procedure, and a Lisp body associated with *proc* exists, then `ocml-control-eval` returns the value obtained by applying the Lisp body to the values obtained by evaluating each $term_i$ in *env*.
12. If *term* has the format (*proc* $term_0$, ..., $term_n$), with n ≥ 0, where *proc* is the name of a procedure, and no Lisp body associated with *proc* exists, then `ocml-control-eval` returns the value obtained by applying the body of *proc* to the values obtained by evaluating each $term_i$ in *env*.
13. In all other cases `ocml-control-eval` signals an error.

4.2 The OCML proof system

4.2.1 Procedure for proving basic goal expressions in OCML

Let's suppose we want to find all solutions to a basic *goal expression*, say *G*, with format (*rel* {*fun-term*}*), where *rel* is the name of a relation and *fun-term* a functional term. In general we might be interested in one, some or all solutions. Therefore the order in which solutions are generated might be important. The algorithm used by the OCML proof system is as follows.

1. If *rel* is not a defined relation, then signal an error. Otherwise initialize *SOL1*, *SOL2*, *SOL3*, *SOL4*, *SOL5* and *SOL6* to the empty set and go to step 2.
2. Retrieve all the assertions present in the current model, whose type (i.e. first element) is *rel*. Match each assertion with *G*. All successful matches, we call this set *SOL1*, provide solutions to *G*. Go to step 3.
3. If a Lisp attachment exists for *rel*, then evaluate it in the Lisp environment to find eventual additional solutions to *G*, say *SOL2*. Go to step 8.
4. If a `:prove-by` proof condition, say *prove-rel-expression*, has been specified for relation *rel*, then compute all solutions to *prove-rel-expression*, say *SOL3*. Each of these is also a solution to *G*. Go to step 8.
5. If a `:iff-def` proof condition, say *iff-rel-expression*, has been specified for relation *rel*, then compute all solutions to *iff-rel-expression*, say *SOL4*. Each of these is also a solution to *G*. Go to step 8.
6. If a `:sufficient` proof condition, say *suff-rel-expression*, has been specified for relation *rel*, then compute all solutions to *suff-rel-expression*, say *SOL5*. Each of these is also a solution to *G*. Go to step 7.
7. If a backward chaining rule has been specified, associated with relation *rel*, then invoke it to find all other solutions to *G*, say *SOL6*. Go to step 8.
8. The set of all solutions to query *G* is obtained by appending the lists *SOL1*, *SOL2*, *SOL3*, *SOL4*, *SOL5* and *SOL6*.

The algorithm shown above provides an operational semantics for the various relation-forming constructs provided by OCML. In particular the following two points should be highlighted.

Assertions inherited through an isa hierarchy are always cached at definition time. This means that they are retrieved at step 2, when the goal is matched against the current set of known facts.

The results returned by non-logical mechanisms such as Lisp attachments and `:prove-by` are only merged with the results obtained by simple assertion-matching (step 2). In other words they are meant to provide efficient proof mechanisms which override those provided by definition-forming options, such as `:iff-def` and `:sufficient`.

4.2.2 Proof rules for non-basic goal expressions

The bullet points below describe how non-basic goal expressions are proven in OCML.

(`and` *A B*). This expression is satisfied if both *A* and *B* can be proven in the current model.

(`or` *A B*). This expression is satisfied if either *A* or *B* can be proven in the current model.

(`=>` *A B*). This expression is satisfied if either *A* cannot be proven, or, if *B* can be proven in each environment which is a solution to *A* .

(`<=>` *A B*). This expression is satisfied if both (`=>` *A B*) and (`=>` *B A*) can be proven.

(`not` *A*). This expression is satisfied if *A* cannot be proven in the current model.

(`exists` *schema-or-var A*). This expression is satisfied if *A* can be proven in the current model.

(`forall` *schema-or-var* (`=>` *A B*)). This expression is satisfied if either *A* cannot be proven, or, if *B* can be proven in each environment which is a solution to *A*..

Thus, the proof mechanism supported by OCML is not complete with respect to first-order logic statements. In particular, disjunctions can only be proved by proving each clause separately and negated expressions are only proved by default.