The CASSANDRA Project:

A 2nd Order CASE Tool in Prolog

Markus Schacher
KnowGravity Inc.
Badenerstrasse 808
8048 Zürich
Switzerland
Phone: ++41-(0)1/434'20'00
Fax: ++41-(0)1/434'20'09
Email: markus.schacher@knowgravity.com

Version: 1.1.2 / 6. April 2000

Abstract

This document describes the software engineering research project "Cassandra" developed at KnowGravity Inc. in Zürich, Switzerland. Cassandra is an assistant that guides software developers through the software development process. It analyzes project information held in one of the many familiar UML-based CASE tools and derives issues to be clarified or suggests the next steps to be done in the project.

Cassandra is a 100% Prolog application, i.e. user interface, knowledge base, persistency and CASE tool access are all implemented in Prolog. This paper discusses important parts of the functionality of Cassandra as well as some Prolog-related implementation issues and experiences.

1. Introduction

1.1 UML and CASE Tools

Since about 3 years the Unified Modeling Language, UML (OMG, 1999) has become the agreed standard for modeling object oriented software systems. It allows modeling a software system by different views (mainly diagrams) complementing each other. The main views (or diagrams) defined by the UML are:

- **Activity diagrams**
  are used for modeling business activities to be supported by the planned IT system
- **Use case diagrams**
  show the main functionalities of the system to be developed (as a black box) from the user perspective
- **Class and object diagrams**
  show classes in the system to be developed, their properties and associations as well as their instances
- **State diagrams**
  show the behavior of a single object in terms of events, states and its (re-)actions
- **Collaboration and sequence diagrams**
  show how multiple objects interact in order to provide a useful functionality to be provided by the system to be developed
- **Deployment and component diagrams**
  model the physical structure of the system to be developed in terms of hardware units and software components.

To simplify the development of these diagrams, a large number of CASE tools (Computer Aided Software Engineering tools) are available on the market. These tools provide facilities to draw at least some of those diagrams and additionally try to keep them consistent based on the fact, that they all should be different views of the same underlying model.

1.2 Cassandra

UML defines a large set of syntactic and semantic rules that a diagram or a set of diagrams must adhere to. On the other hand, UML says very little about how these diagrams should be used to develop a (highly complex) real-world application. Over years, KnowGravity Inc. and its partners developed a number of techniques as well as the know-how required to elaborate specifications and architectures using UML notation (i.e. its diagrams). So far, this know-how has been taught in software engineering courses and applied in large IT projects coached by KnowGravity's consultants. However, the high demand for coaching from KnowGravity's clients was faced with the limited availability of consultants at KnowGravity Inc.

This situation led to the idea, to make the consultant’s know how wider available through a knowledge-based system, called **Cassandra** (**C**assandra - an **A**ssistant for **S**ystem **S**pecification **AND** **R**equirements **A**nalysis). It is an experimental software system to support software engineering based on techniques from the field of artificial intelligence. Cassandra has been implemented using the programming language WIN-PROLOG (LPA, 1997) running on a standard PC/Windows platform.

1.3 Related approaches

Instead of providing various editors to draw UML diagrams (as conventional CASE tools do), Cassandra supports software developers in the usage of any conventional CASE tool in order to guide them though the process of developing an application. However, some of the more advanced conventional CASE tools also provide some kinds of assistants or wizards to support the user in repetitive tasks. Others include process mentors or guides that are online documentations of a generic software development process. In contrast to these approaches, Cassandra

- is independent of any specific CASE tool
- provides guidance based on the actual state of a project or model in a CASE tool
- acts as a very generic container for software engineering know how.

Currently, not many similar approaches are known. On http://www.asplake.demon.co.uk, a tool called **Validator** is published, that provides similar pattern analysis as Cassandra does for developing data models.

2. Cassandra

2.1 Architecture

Cassandra consists of a basic platform application that can be extended by several extension modules at runtime (see figure 1). These modules are added to the Cassandra core system by simply copying its files into Cassandra's main directory without any need for reconfiguration or even recompilation.

The size of the Cassandra application is currently about 7'000 lines of Prolog code, of which about 3'000 lines are generic and thus highly reusable (module management, license management, GUI handling, internationalization, and timing).

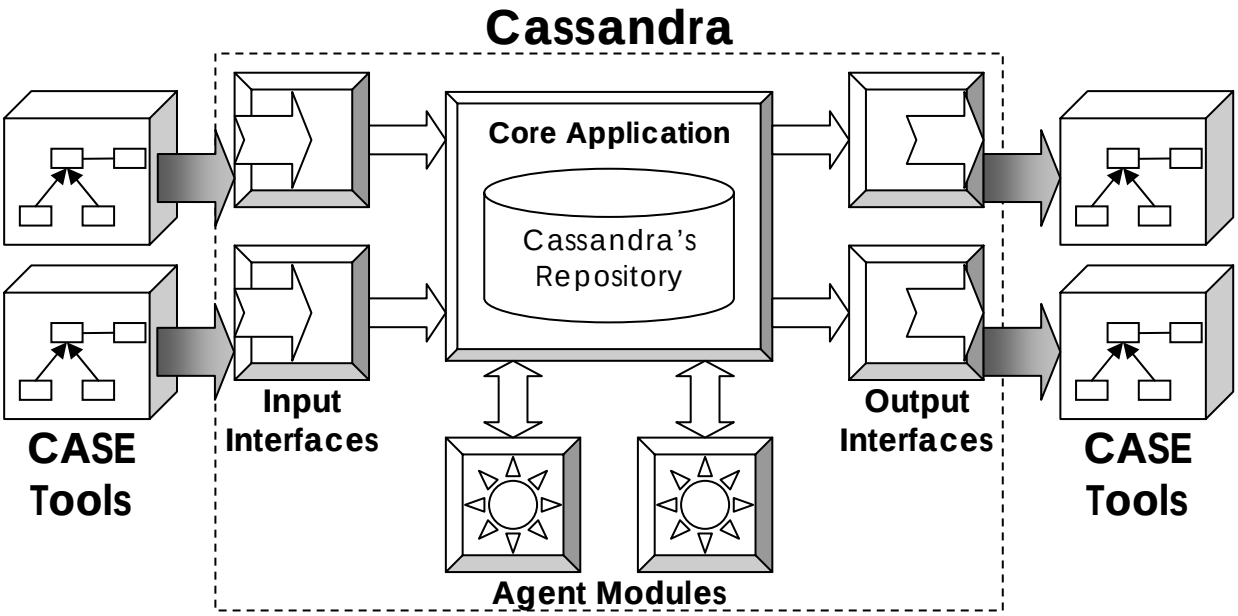

Figure 1: Cassandra's Architecture

- **The Core Application**
  The Core Application provides some generic infrastructure functionality used by the individual extension modules. At the core of it is Cassandra's active Repository that stores project information imported from various CASE tools.
- **The Input Interfaces**
  An Input Interface is an extension module that is able to import project information from an external CASE Tool into Cassandra's Repository. An Input Interface provides its own user interface that is plugged-in into the Core Application.
- **The Agent Modules**
  An Agent Module is an extension module that implements some useful software engineering functionality based on the information stored in Cassandra's Repository. It provides its own user interface that is plugged-in into the Core Application.
- **The Output Interfaces**
  An Output Interface is an extension module that is able to export project information to an external CASE Tool based on the decisions taken by an Agent Module. Output Interfaces too provide their own user interface that is plugged-in into the Core Application.

2.2 The User Interface

Cassandra can either be started from within Windows or directly from within some CASE tools. In any case, Cassandra's main dialog as shown in figure 2 is displayed.

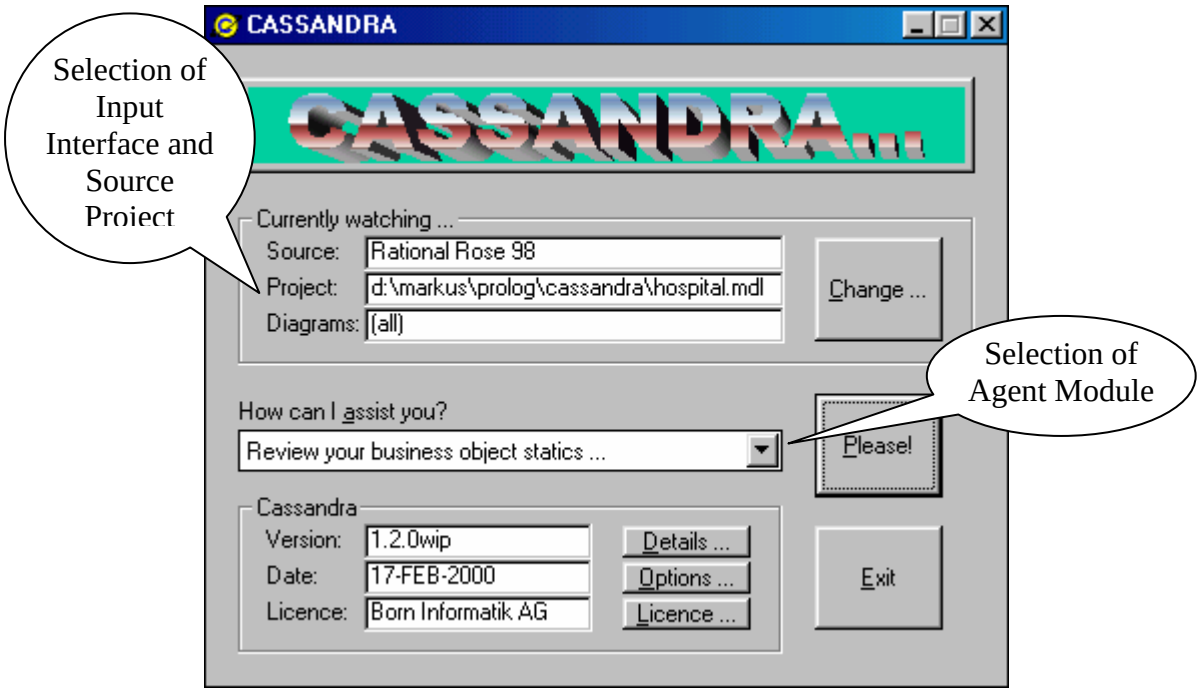

Figure 2: Cassandra's Main Dialog

Basically, this dialog allows doing two things: to select the project to be investigated (including the selection of the Input Interface, i.e. the CASE tool) and to select the Agent Module that provides the actual investigation or assistance.

Currently, Input Interfaces for the following CASE tools are available (no Output Interfaces have been developed so far):

- Artisan RTS V3.0 (XIARTRTS30)
- Select Enterprise V5.2 (XISELE52)
- Select SSADM V4.0 (XISELS40)
- Rational Rose 98 (XIROSE98)

All these Input Interfaces access the corresponding CASE Tools via OLE automation (a standard on the Windows platforms for inter-application communication). In addition, an Input Interface to the Repository's native file format is provided.

When the corresponding project data is available, the user can currently select from one of the following Agent Modules that act on the project:

- **XARVBOST**
  Performs a domain-level review of the static structure of the business object model.
- **XARVUCST**
  Performs a domain-level review of the static structure of the use case model.
- **XADWH**
  Generates a first-cut data warehouse design out of the model of an operational IT-system.

By now, the result of such an investigation is presented as a textual report. Later, it is intended to provide a more interactive version of these Agent Modules so that they communicate interactively with the user.

2.3 The Repository

Cassandra's Repository can be regarded as an object oriented database system implemented in Prolog. Its structure is defined by a meta model that defines the software engineering concepts Cassandra is dealing with (e.g. classes, states, use cases and so on). Since every CASE tool is based on a (at least) slightly different definition of those concepts, Cassandra's meta model represents our (KnowGravity Inc.) unified view of software engineering that can be mapped on any CASE tool supported by Cassandra.

The Repository's meta model is defined by a set of simple Prolog facts that allow the specification of meta classes, their attributes, associations and operations as well as inheritance associations among them. As an example, figure 3 shows a very simple (non UML) meta model describing a fragment for representing a data model.

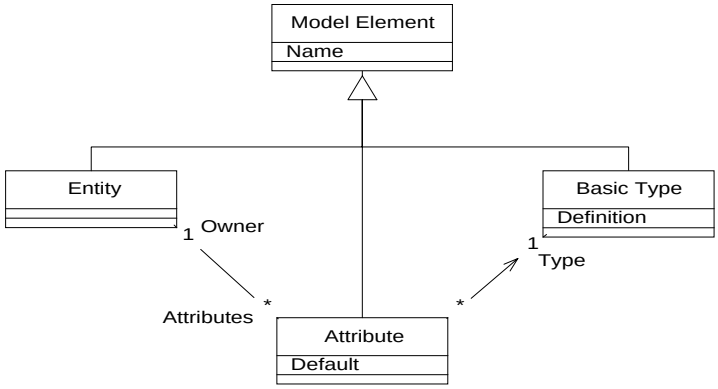

Figure 3: A simple meta model

The Repository definition of the meta model in figure 3 is shown in listing 1. A meta model definition may consist of the following elements:

1. Meta model name and version declaration
2. Basic data type declarations
3. Meta type declarations
4. Subtype/supertype association declarations
5. Partition declarations

A meta object type declaration usually consists of a set of property declarations. Each property may be an attribute or reference to another meta object type. Such a reference declaration may either be declared as a simple `item` (default), `set` or `bag`.

```
mm_version('A very simple meta model', '0.1.0', ['0.1.*']).

name(Name) :- atom(Name).

rep_type(model_element, [name(name)]).
rep_type(entity, [attributes(ref(attribute), set)]).
rep_type(attribute, [default,owner(ref(entity)),type(ref(basic_type))]).
rep_type(basic_type, [definition]).

rep_isa(entity, model_element).
rep_isa(attribute, model_element).
rep_isa(basic_type, model_element).

rep_partition(structure, [entity, attribute]).
```

Listing 1: The definition of the simple meta model

The meta objects declared in this way may be manipulated by a set of built-in access predicates. These can be classified as follows:

- predicates to create, find, inspect, and delete meta objects
- predicates to manipulate attributes
- predicates to manipulate associations
- predicates to initialize, load and unload the Repository.

In addition to the predefined object access predicates, user-defined operations may be declared for a meta object type. User defined operations are defined as ordinary Prolog predicates that have the meta object type and the object ID as their first two arguments.

Any object operation (built-in or user-defined) may be invoked using the dot operator in one of the following forms:

`ObjectId.Operation(Parameters)` or `(ObjectType, ObjectId).Operation(Parameters)`

Based on the simple meta model shown above, the program in listing 2 instantiates a system model containing one single entity `customer` with attributes `name`, `birthday` and `address`.

```
test :-
  rep_init,

  % create objects
  (basic_type, Text).new([name=text]),
  (basic_type, Date).new([name=date]),
  (entity, Customer).new([name=customer]),
  (attribute, Name).new([name=name]),
  (attribute, Birthday).new([name=birthday]),
  (attribute, Address).new([name=address]),

  % set some attributes
  Text.set_prop(definition='Any text of any length'),
  Date.set_prop(definition='Any date in format DD-MM-YY'),
  Name.set_prop(default='(no name)'),
  Address.set_prop(default=''),

  % create unidirectional associations
  Name.set_prop(type=Text),
  Birthday.set_prop(type=Date),
  Address.set_prop(type=Text),

  % create bi-directional associations
  link_md(Customer, attributes, Name,     owner),
  link_md(Customer, attributes, Birthday, owner),
  link_md(Customer, attributes, Address,  owner),

  % display objects
  Text.show,
  Date.show,
  Customer.show,
  Name.show,
  Birthday.show,
  Address.show,

  % delete created objects
  Text.del,
  Date.del,
  Customer.del,
  Name.del,
  Birthday.del,
  Address.del.
```

Listing 2: An example predicate that uses the simple meta model

One of the interesting features of Cassandra's Repository is that it may also lookup objects using the same syntax as shown above. For example the call

```
?- OID.get_prop(name=abc).
```

returns all objects that have "`abc`" as their name, irrespective of which type they are, whereas the call

```
?- (attribute, OID).get_prop(name=abc).
```

returns all objects that are of type "`attribute`" that have "`abc`" as their name.

2.4 Cassandra's Meta Model

However, the actual meta model of Cassandra is much more complex than the one shown in figure 3. Figure 4 shows the part of Cassandra's current meta model that represents an extended UML class model.

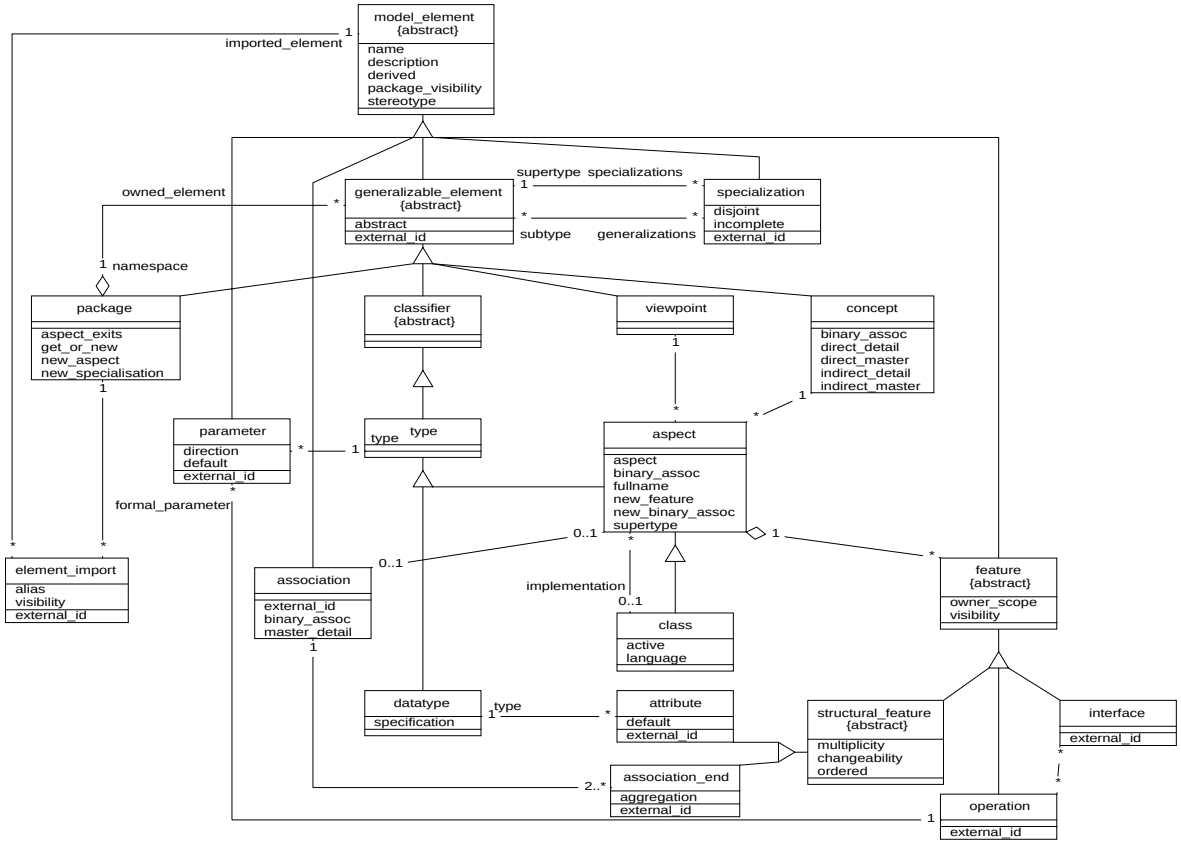

Figure 4: The fragment "Object Statics" of the Cassandra meta model

In addition to the meta model fragment shown in figure 4, Cassandra's meta model is partitioned into the following areas:

- **Object Dynamics**
  Covers state diagrams for individual objects
- **Object Interactions**
  Covers collaboration and sequence diagrams
- **External Design**
  Covers business models and use case diagrams
- **Implementation**
  Covers component and deployment diagrams
- **Process**
  Covers activities of the development process as well as its results.

2.5 The XARVBOST Agent

As an example, Cassandra's Agent Module XARVBOST is explained in more detail. This Agent Module provides a pattern-based review of the static structure of any given business object model. There are some common structures (or patterns) known on objects (classes) and associations that raise domain level questions (see figure 5 for two typical patterns).

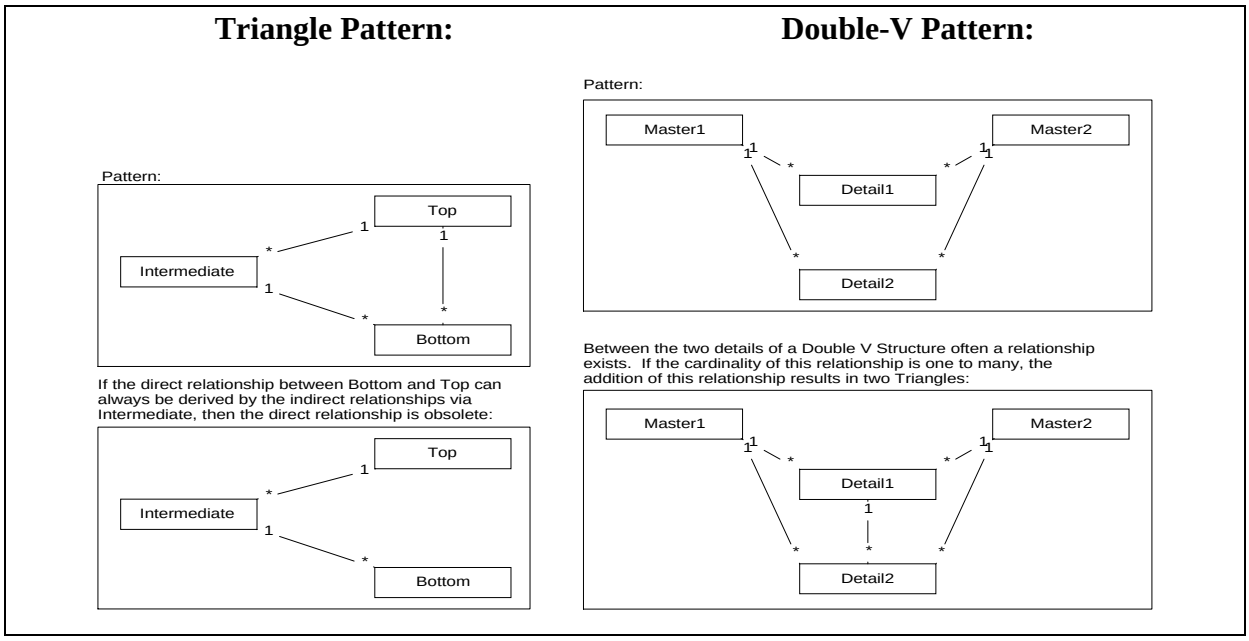

Figure 5: Triangle and Double-V Patterns

These structures are purely geometric shapes that raise business level questions. A business object model can be searched for these patterns and the resulting questions can be asked to the user. Typical questions derived from such patterns are whether an object or an association is missing or whether an object or association is obsolete.

Beside the basic forms of these patterns as shown in figure 5, more complex forms are known that include indirect or special associations as well as inheritance associations between objects. These forms still have the same characteristics as Triangle and Double-V patterns explained above but are much harder to find. Further details about the technique to apply these structures can be found in our course materials (KnowGravity, 1997).

When the simple sample business object model shown in figure 6 is presented to Cassandra for review, in a first step its structure is transferred from the CASE tool into Cassandra's Repository (via the corresponding Input Interface).

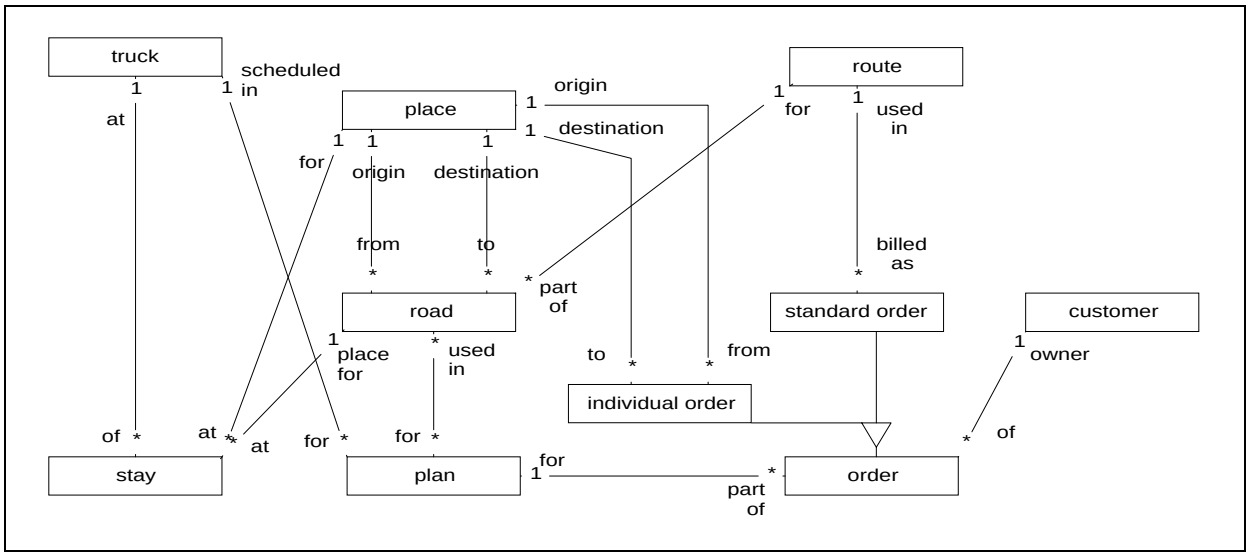

Figure 6: An example Business Object Model

Then, the options dialog (figure 7) is shown to the user. This options dialog allows the selections of patterns to be searched as well as the specification of the desired output type for the analysis. Two different types of reports are available:

- The **Domain Expert Report** provides the domain level questions derived from the patterns found in the model in a natural language selectable by the user.
- The **Analyst Report** presents the patterns found by expressing these patterns in terms of their geometric structure comprising objects and associations among them.

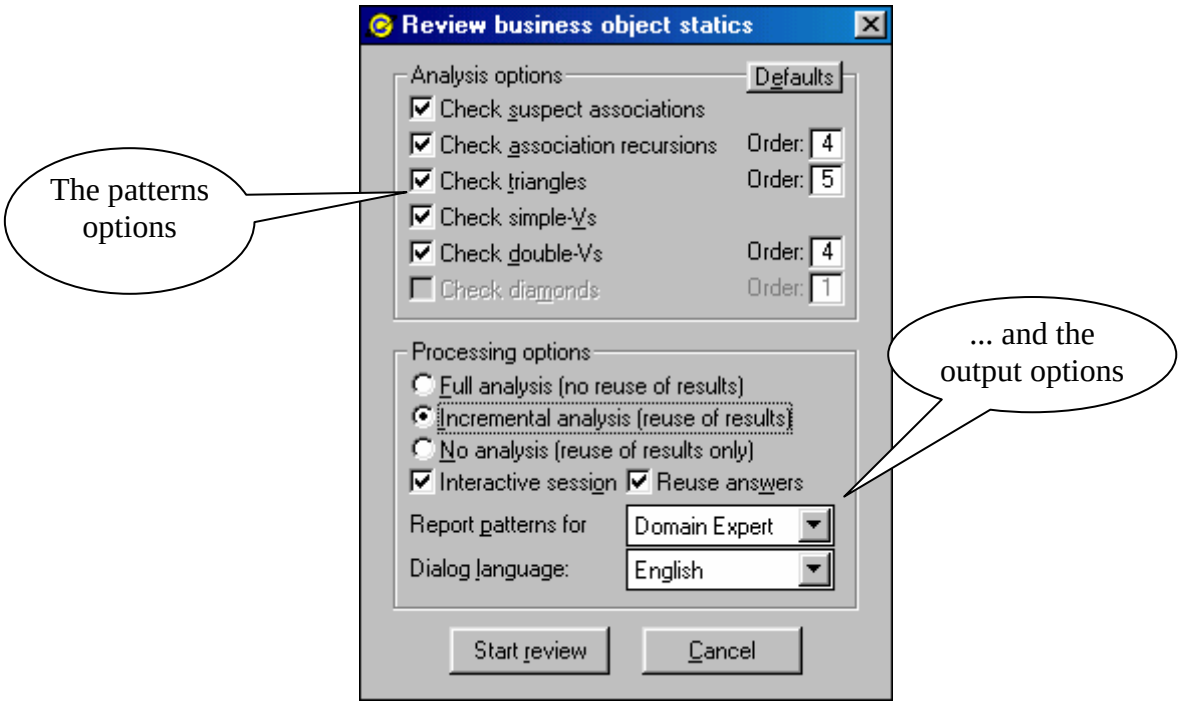

Figure 7: The XARVBOST Main Dialog

Figure 8 shows the Domain Expert Report produced after analyzing the example business object model shown in figure 6.

```
XARVBOST: Review of business object statics (Domain Expert)
============================================================
CASSANDRA V1.1.0 / 26-OCT-99
XARVBOST  V0.1.2 / 26-OCT-99

  Source         : File
  Project        : d:\prolog\cassandra\deployment v1-1-0\fast-english.rep
  Directory      : d:\prolog\cassandra\deployment v1-1-0\
  Last change    : 27.10.1999 / 17:44: 6.00

Potentially redundant associations
----------------------------------
If the statement "stay at one place" has the same meaning as the
  statement "stay at one road to one place",
  the direct association between "place" and "stay" is redundant.
  ... or if it has the same meaning as the statement
  "stay at one road from one place".

Potentially missing associations
--------------------------------
Is there an association missing between "individual order" and "stay"?
Is there an association missing between "standard order" and "stay"?
```

Figure 8: A Domain Expert Report

3. Some Implementation Concepts

This section focuses on some issues that might be specifically of interest to Prolog and/or more specifically WIN-PROLOG programmers.

3.1 Module Management

Cassandra has been built in a very modular way. These modules are managed by a module handling system developed earlier by the author. This module handling system provides the following features:

- Module version, dependency and compatibility management
- Multi-language (human languages) support
- Prolog operator declarations
- Compile-time and/or runtime module loading

In this module handling system, a module may consist of one to three individual files:

- The main file contains the predicate implementations provided by the module and may either be in source form (*.PL) or compiled object form (*.PC).
- The optional message file (*.msg) contains a number of Prolog facts that declare message texts used by the module in one or more international languages
- The optional operator file (*.opr) declares any operators needed (or provided) by the module.

As an example, Cassandra's Repository is a module composed of the files `REPOSITORY.PC`, `REPOSITORY.MSG` and `REPOSITORY.OPR`. Figure 9 shows the Module Details Dialog of Cassandra that displays details about the currently loaded modules.

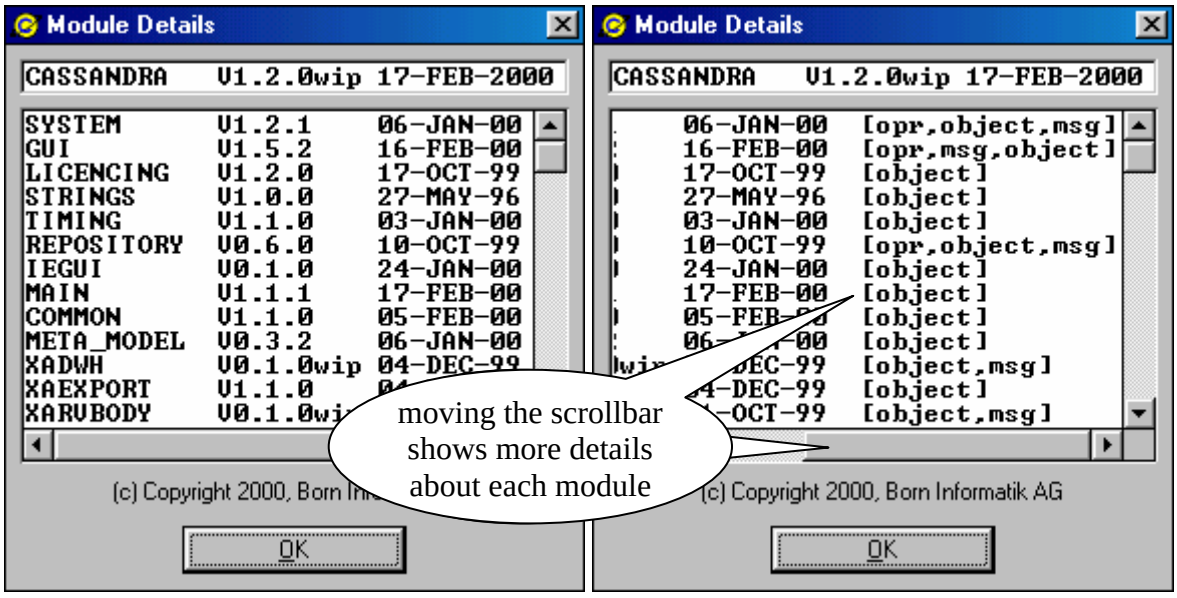

Figure 9: The Module Details Dialog

In order to provide an automatic compatibility check among different modules loaded by an application, each module must declare the following information:

- its name, author and creation date
- its current version number
- a (possibly empty) list of version numbers to that the current version of the module is backward compatible
- a (possibly empty) list of other modules that are used by this module, optionally including the required version number of the required module.

A module declares this information by calling the predicate `mv/6`. The arguments of `mv/6` are as follows:

```
mv(Module, Version, Date, Author, BackwardCompatibility, RequiredModules)
```

This declaration (i.e. the invocation of mv/6) must be performed as soon as the module has been completely loaded, which can be ensured by declaring it using the WIN-PROLOG built-in predicate `initialization/1`. Listing 3 shows an example declaration of a module `DEMO` including all elements listed above.

```
:- initialization(mv(demo, '2.1.3', '25-NOV-99', 'M. Schacher', ['0.3.*', '1.0.*'],
                     [system('1.2.0'), gui, strings, licencing('1.1.3'), common])).

... the actual module code starts here
```

Listing 3: A module declaration

The main module of an application actually loads all required modules by using the predicate `load_modules/2`. This can happen either at compilation time of the main module (static loading) or at runtime of the application (to dynamically load additional modules). See Listing 4 for an example.

```
:- load_modules([licencing, strings, gui, timing], [path('c:\prolog'), log]).
```

Listing 4: Loading modules

In any case, when all modules have been loaded, `load_modules/2` performs a compatibility check among all loaded modules based on their version information. Any

incompatibilities or missing modules are reported as warnings or error messages to the user (see figure 10 as an example).

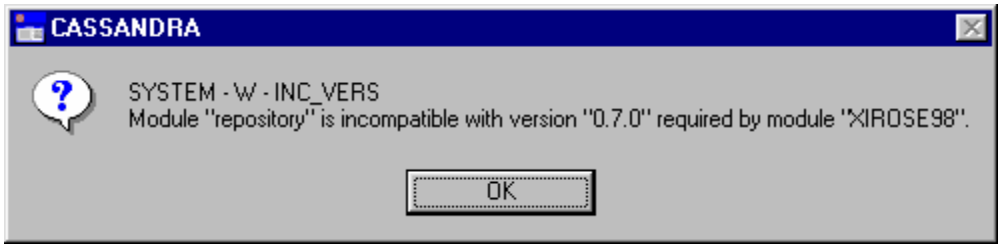

Figure 10: An incompatibility warning

Cassandra uses dynamic loading of modules at runtime to allow an easy addition and/or removal of Input Interfaces and Agent Modules. At startup time, any module that is found in Cassandra's main directory and that has a name starting with the letter "X" (e.g. "XIROSE98" or "XARVBOST"), is automatically loaded and checked for compatibility against all other modules by using `load_modules/2`.

3.2 Extendable User Interface

Cassandra has been implemented completely in WIN-PROLOG, including its user interface. Since Cassandra dynamically loads extension modules at startup time, at least some parts of the user interface depend on the currently loaded modules. Figures 11 and 12 show such an effect in case of different Input Interfaces. Depending on the selected Input Interface (source), different options apply to that Input Interface.

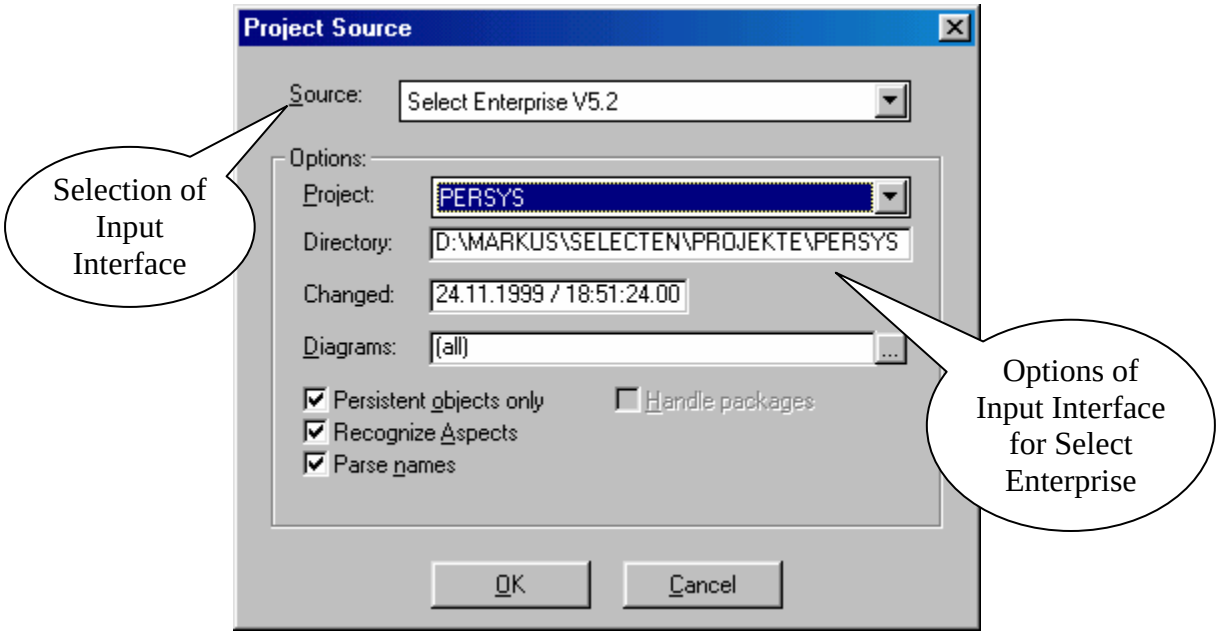

Figure 11: The Source Selection Dialog for Select Enterprise

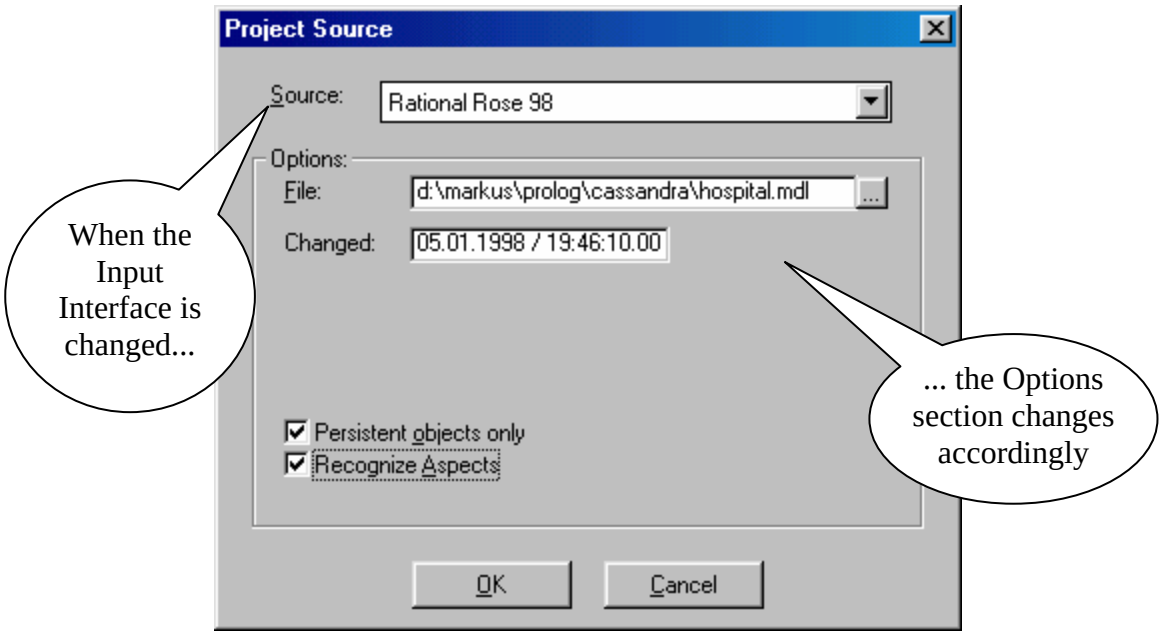

Figure 12: The Source Selection Dialog for Rational Rose

However, the user interface elements in the current options group are part of the actual Input Interface and are thus implemented in the corresponding (dynamically loaded) module. This means that the implementation of the dialog shown above is distributed among several modules:

- The dialog frame including the combobox to select the Input Interface is implemented in Cassandra's Core Application (also called the master module).
- All items inside the options group are implemented in one of the dynamically loaded Input Interface modules (also called the client module).

To achieve this cooperation among different modules sharing one single dialog, a protocol among modules has been defined that includes

- the client area of a master dialog that may be used by a client module
- the activation of a client module, that causes the creation and handling of the client-specific dialog items
- the deactivation of a client module, that causes the destruction of the client-specific dialog items.

Listing 5 shows a fragment of the dialog handler that processes the event of changing the selection of the Input Interface combobox in the master dialog: the old Input Interface is deactivated by the call `xdlg_CASS_source(XI_old, off)` and then the new Input Interface is activated by the call `xdlg_CASS_source(XI_new, on)`.

```
xdlg_CASS_source_handler((xdlg_CASS_source,590), msg_select, _, _) :-
  !,
  get_tstate(cass_main, current_xi, XI_old),           % determine the old client
  xdlg_CASS_source(XI_old, off),                       % ... and turn in off
  lbx_get_selected((xdlg_CASS_source,590), Source, _), % read the combobox
  xi_id(XI_new, Source),                               % ... get the new client id
  set_tstate(cass_main, current_xi, XI_new),           % ... remember it
  xdlg_CASS_source(XI_new, on).                        % ... and turn it on
```

Listing 5: Client dialog activation and deactivation

3.3 Pattern Analysis

One of the more challenging tasks has been the development of the pattern analysis described in section "2.5 The XARVBOST Agent". The pattern search algorithms are implemented as highly declarative operations of the meta model objects (see also figure 4). For example, listing 6 shows a slightly simplified version of the main predicate that searches all associations playing the role of the direct association in a triangle.

```
search_triangles :-
  MaxOrder=9,
  (association, Assoc).triangle(Direct, Indirect, MaxOrder, Order),
  write((Assoc, Direct, Indirect, Order)), nl,
  fail.
search_triangles.
```

Listing 6: Searching for triangles

However, the complexity of the search algorithm is buried inside the user-defined operation `triangle/4` of the `association` object. Listing 7 shows some details of this operation. It is based on the user-defined operations `master_detail/2` of class `association` and `indirect_detail/6` of class `concept`. `master_detail/2` returns the master and detail `concept` of an association including the corresponding cardinalities (zero to one, zero to many, etc.). `indirect_detail/6` returns any `concept` that is indirectly related to the `concept` under investigation via a path of associations.

```
triangle(association, Assoc, Direct, Indirect, MaxOrder, Order) :-
  (association, Assoc).master_detail([TopC, MC], [BottomC, _]),
  Direct = [node(TopC), link(Assoc), node(BottomC)],
  (concept, TopC).indirect_detail([_, MC], Indirect, _, BottomC, Direct, MaxOrder),
  length(Indirect, NIndirect), Order is (NIndirect-3)/2, Order>0.
```

Listing 7: The operation triangle/4 (simplified)

Passing the `Direct` path to `indirect_detail/6` ensures that the direct association will never be part of the indirect path. Further, an important condition is that the cardinalities of both direct and indirect paths must be the same, which is achieved by the shared variable `MC` for master cardinality). However, the real work is done in the two operations `master_detail/2` and `indirect_detail/6`. Since associations are not attached to `concept` but to `aspect` (see figure 4 again) these two predicates are responsible to provide this transformation towards the meta model. Listing 8 finally shows how this is actually done in the case of `master_detail/2`. This predicate also determines the correct master and detail concepts depending on the cardinalities of the association.

```
master_detail(association, Assoc, [MasterC, MasterCard], [DetailC, DetailCard]) :-
  (association, Assoc).binary_assoc([Aspect1, Role1, Card1], [Aspect2, Role2, Card2]),
  ((Card1=[1, 1], Card2\=[1, 1]) ->            % *** Aspect1 is Master
    MasterA=Aspect1, MasterCard=Card1, DetailA=Aspect2, DetailCard=Card2
  ; ((Card1\=[1, 1], Card2=[1, 1]) ->          % *** Aspect2 is Master
      MasterA=Aspect2, MasterCard=Card2, DetailA=Aspect1, DetailCard=Card1
    ; fail)),                                  % *** Master/Detail not decidable
  (aspect, MasterA).get_prop(concept=MasterC),
  (aspect, DetailA).get_prop(concept=DetailC).

binary_assoc(association, Assoc, [Aspect1, Role1, Card1], [Aspect2, Role2, Card2]) :-
  (association, Assoc).get_prop(association_ends=[AE1, AE2]),
  (association_end, AE1).get_props([aspect=Aspect1, name=Role1, multiplicity=Card1]),
  (association_end, AE2).get_props([aspect=Aspect2, name=Role2, multiplicity=Card2]).
```

Listing 8: The operation master_detail/2 (simplified)

As can been seen from the explanations above, searching a pattern involves many levels of Repository predicates. In Cassandra, these levels are distributed among several modules:

- The specific operations (like `triangle/4`) are implemented in the Agent Module XARVBOST that actually does the pattern analysis.
- The generic operations (like `master_detail/2` and `binary_assoc/2`) are implemented as generally reusable operations in the module that represents Cassandra's meta model.
- The Repository operations (like `get_prop/1`) are part of Cassandra's Repository, which is a module of its own.

This finally means that the actual program performing the pattern analysis is composed at startup time of Cassandra. In other words: Since the Agent Modules are loaded at Cassandra's startup time, the actual capabilities of each meta object type are built dynamically and depend upon the currently loaded extension modules!

4. Experiences

4.1 OLE Automation

When the development of the first ancestors of Cassandra was started, there was no way in WIN-PROLOG to communicate with another application via OLE automation. As a consequence, the author started to develop a DLL in C++ that adds this capability into WIN-PROLOG via WIN-PROLOG's extensibility interface.

After some discussions with LPA (the developer of WIN-PROLOG), it turned out that such a capability could also be of interest to other developers using WIN-PROLOG, for example to control any Microsoft Office application (e.g. Word) from within a Prolog program. For that reason, LPA integrated an OLE automation interface into the core of WIN-PROLOG and made it generally available. It is based on the original concepts of the author's OLE DLL, but it is today much more robust than the original one.

4.2 Module Management

The module management subsystem described in section "3.1 Module Management" has been first implemented by the author on another Prolog implementation (Daum, 1989) and then ported to LPA's WIN-PROLOG. This raised two difficulties:

- WIN-PROLOG identifies files currently loaded by their primary name only, ignoring the file extension. This caused some trouble in separating modules into code files, message files and operator files.
- There is no (official) way to hide atoms in a module of WIN-PROLOG. This bears the danger that two atoms with the same name may be defined in different modules. Especially when more than one programmer is involved, this is not unlikely to happen. On the other side there are situation where a specific atom is deliberately needed across more than one module. It turned out, that in WIN-PROLOG depending on how such an atom has been created, atoms of the same name do not unify! The problem is not so much specific to WIN-PROLOG: There needs to be a clear standard that defines the exact semantics of modules in a Prolog application.

4.3 User Interface

WIN-PROLOG provides a set of built-in predicates and a simple dialog editor that allows to build all components of a simple graphical user interface without the need of any other programming language. Using these capabilities the author was able to build Cassandra as a standalone application that is 100% implemented in Prolog. However, more advanced GUI-elements such as tabbed dialogs are not directly available.

4.4 Repository Implementation

Cassandra's Repository can be considered as a highly declarative object oriented language. It turned out (as in several projects before) that Prolog is extremely powerful to implement new programming languages having their own semantics and syntax.

Especially the creation of languages with a highly declarative character can be implemented very elegantly in Prolog. As it has been shown in section "3.3 Pattern Analysis", the object operation `triangle/4` is purely declarative: it simply says that a given association plays the role of a direct association in a triangle. It can be used with any of its argument free or instantiated (except `MaxOrder` that must be instantiated).

Even the arguments identifying the association can be left free: the following call returns any direct `association` in a triangle with a maximum order of 9:

```
?-(association, Assoc).triangle(Direct, Indirect, 9, Order)
```

4.5 Performance

The performance of Cassandra's Repository has been extensively tested. Below, some of the results on a Pentium III/500 machine are summarized:

- Object creation: about 0.12ms (for simple objects) to about 2.4ms (for complex objects)
- Object deletion: about 0.35ms (for simple objects) to about 2.79ms (for complex objects)
- Property setting: about 0.22ms (for local properties) to about 0.49ms (for inherited properties)
- Operation invocation: about 0.009ms (for early binding) to about 1.43ms (for late binding)
- Object lookup: about 0.04ms (for simple lookups) to more than 14.88ms (for complex lookups)

These measures do not look too bad. However, since some very complex algorithms are running on top of these Repository predicates (such as the pattern analysis of XARVBOST), they can never be fast enough.

The sample business object model shown in figure 6 requires on a Pentium III/500 machine about 5½ seconds to find all triangles and about 15 seconds to find all Double-Vs. Unfortunately the time required for such pattern analyses does not increment linearly with the number of business objects: Second order Triangle analysis for a model containing 88 business objects lasted about three hours on a Pentium III/500 and 9½ hours for a first-order Double-V analysis.

4.6 By-products

During development of Cassandra, several generally useful by-products have been developed. First of all, Cassandra's Repository is becoming a product of its own. It can be used as a general object oriented extension for any Prolog application. Furthermore, it may be used as a database capable of easily simulating an application modeled in a CASE tool.

Another by-product of Cassandra is a handy regression tester mainly used during the development of the Repository (see figure 13).

Figure 13: The Regression Tester

It is fully integrated into WIN-PROLOG's development environment and provides the following features:

- recording (console) output of predicates to be tested
- replaying of tests and automatic comparison of output against pre-recorded output
- manual comparison of replayed output against pre-recorded output
- storage of tests and test results in a persistent database
- individual tests can be combined into test groups for batch testing
- non-modal user interface can be kept open in the development environment.

In addition, several general purpose module have been developed that are reusable outside Cassandra. These are mainly:

- **SYSTEM**
  Implements module management, multi-language text handling channel-based I/O and some general purpose predicates.
- **GUI**
  Implements some reusable dialogs and some general GUI related predicates.
- **STRINGS**
  Implements some predicates to process strings and atoms.
- **TIMING**
  Implements some predicates to perform time and date calculations.
- **LICENCING**
  Implements a flexible mechanism for licensing applications.

5. Summary and Next Steps

Cassandra is still a software engineering research project. Currently, it is mainly used by our own consultants to simplify their jobs working on real projects. We still consider Cassandra as our personal weapon that differentiates us from our competitors. However, the recent version of Cassandra has been given to some of our customers for a very restricted beta test. It is planned to publish a public domain version of Cassandra on our web site (KnowGravity, 2000) later this year.

Within the next few months, it is planned to implement the following Agent Modules to expand Cassandra's software engineering capabilities:

- **XAAPI**
  Provides some support in application partitioning and integration.
- **XAMODBO**
  Actively supports the user in identifying and modeling business objects.
- **XAMODED**
  Actively supports the user in identifying and modeling business activities and use cases.
- **XAPGIS**
  Implements an active process guide for information systems development.

It is further planned to integrate a multi-language rule-based inference engine into Cassandra. This would allow an improved support for project situation analysis and diagnosis. Finally, in the long terms, a simulation engine should allow to execute a specification in UML without the need of one single line of code. A similar tool based on structured analysis has been developed nearly ten years ago by the author (Schacher, 1991).

The ultimate vision for Cassandra is that it may some day replace our consultants completely. We then can send Cassandra to help our customers with their project instead we go by ourselves. The only remaining question then is: How do we earn our money?

6. References

(KnowGravity, 1997) KnowGravity Inc.: UML course material, 1997-1999

(KnowGravity, 2000) Web site of KnowGravity Inc.: www.knowgravity.com

(Daum, 1989) Dr. Berthold Daum: Salix Prolog 2 Professional, Heim Verlag, 1989

(LPA, 1997) Logic Programming Associates Ltd.: WIN-PROLOG (user documentation) , V3.500, 1997

(OMG, 1999) OMG: Unified Modeling Language Specification, Version 1.3, June 1999

(Schacher, 1991) M. Schacher: SESAM Benutzerhandbuch, Version 0.6, 1991