The Mandarax Manual

Jens Dietrich

Institute of Information Sciences & Technology
Te Kura Putaiao o Hangarau-a-Mohiotanga
Massey University
Palmerston North
New Zealand

j.b.dietrich@massey.ac.nz

Version: 8-Dec-03

made with:

OpenOffice.org 1.1

Table of Content

Acknowledgments

The author would like to thank the following people and organizations: **SourceForge.net** for doing such a great job, **Bauer & Partner Software and Consulting** (my former employer) for sponsoring the mandarax web site, the **Polytechnic of Namibia** (another former employer) and **Massey University NZ** (my current employer) for supporting the mandarax research project, and the following individuals for their contributions to the mandarax project: **Jochen Hiller** (Bauer & Partner), **David Li** (Digital Sesame), **Chen Jianbo** (China Telecom Guangzhou R & D Center), **Alex Kozlenkov** (City University London), **Hans-Henning Wiesner** (Bauer & Partner), **Wang Liangbin** (Donghua University in Shanghai/China) and **Robert Searle** (Platinum Architecture Group).

Synopsis

Mandarax is an open source[1] java library for inference rules. This includes the representation, persistence, exchange, management and processing (querying) of rule bases. The main objective of mandarax is to provide a pure object oriented platform for rule based systems.

Installation

The Mandarax Distribution

Mandarax is distributed as a zip archive. The current version of this zip file can be downloaded from `sourceforge.net/projects/mandarax`. First unzip this file into a folder (for instance, named `mandarax_home`) (`c:\mandarax` for windows or `/home/myself/mandarax` for Unix/Linux).

The mandarax folder has the following file structure:

Folder/File	*Content*
`build.bat`	Windows build script
`build.sh`	Unix build script
`build.xml`	ant build script
`bin/`	contains apache ant (build tool)
`build/`	Contains the compiled mandarax classes and libraries. If this folder is missing or empty, the build still has to be performed (using `build.bat` or `build.sh`, respectively)
`config/`	Miscellaneous files including a JIndent configuration file and a manifest template used for building the jar files
`docs/`	Contains the manual, a `TODO` and a `ChangeLog` text file.
`lib/`	3rd Party Libraries (as jar files) mandarax needs.
`src/`	The mandarax source code.
`tmptestdata/`	This folder will be used to store files generated by mandarax test cases.

Table 1 The Structure of the Mandarax Distribution.

Building Mandarax

[1] GNU Lesser General Public License

Building mandarax will compile the source code, build the jar files and create the javadocs. To build mandarax, open a shell window or dos command prompt, cd to `mandarax_home` and run `build.sh` (Unix) or `build.bat` (Windows). This will compile the classes (`build/classes`), build the jar files (`build/lib`) and the `javadoc` documentation (`docs/api`).

Troubleshooting

Problem: The JDK cannot be found.
Solution: Check your JDK installation. In particular, make sure that the environment variable JAVA_HOME is set correctly and points to the folder where the JDK is installed (and not to the bin sub folder).

Problem: The build fails with a message indicating that the JDK version is wrong.
Solution: Check your JDK installation. In particular, make sure that the environment variable JAVA_HOME is set correctly and points to the folder where a the JDK version 1.4 or better is installed. Verify this with the command `java -version`.

Problem: The build fails and a message `“ANT_HOME must be set first”` is displayed.
Solution: Set the `ANT_HOME` environment variable. To use the ant installation that is part of the mandarax installation, run the following command (`mandarax_home` must be replaced by the real path):

```
set ANT_HOME=mandarax_home\bin\apache-ant-1.5.2
```

For Linux, read the documentation of your shell to find out how to set environment variables. You can also set a global variable. For the windows platform, the configuration form is located in `Configuration – System - Advanced`.

Mandarax Executables

Mandarax is a class library, so there is no mandarax main class. However, there are some classes in the `test` and `org.mandarax.example` packages that are executable. In order to execute them, all libraries in the `lib` and in the `build/lib` folder must be in the class path.

Logical Objects

Rules, Facts and Terms

Java interfaces and classes represent the various elements in logical expressions. We explain these objects using an example. Consider the following sentence:

> If the turnover of a customer was more than 100$ in the year 2002 then the customer gets a discount of 5%.

This sentence can be decomposed as follows:

```
If the turnover of a customer was more than 100$ in the year 2002 then the customer gets a discount of 5%
        If the turnover of a customer was more than 100$ in the year 2002
                more than
                the turnover of a customer in the year 2002
                        the turnover
                        a customer
                        in the year 2002
                100 $
        the customer gets a discount of 5%
                gets discount
                the customer
                5 %
```

The top – level construct is called a *rule*. A rule associates one or many prerequisites with one *conclusion*. Sometimes the prerequisites are also called the *body* of the rule, while the conclusion is called its *head*. The meaning of a rule is simple: whenever all prerequisites are true, the conclusion is true as well. Usually the prerequisites of a rule are connected using *"and"*, meaning that all prerequisites must be satisfied. Prerequisites can also be connected by *"or"*. This means that the conclusion is true if at least one of the prerequisites is true.

The prerequisites and the conclusion are *facts*. Facts can also occur standalone, for instance "Max spent 42 $ in 2002". The facts themselves consist of *terms* and a *predicates* associating those terms. In the example above, "more than" and "gets discount" are both predicates, while "100 $", "5$", "a customer" and "the turnover of a customer in the year 2002" are terms. Speaking OO, terms represent objects. Predicates on the other hand represent a relationship between terms.

There are three different kinds of terms: *constant terms*, *variable terms* and *complex terms*. Constant terms are more or less concrete objects like "5 %" or "100$ " in the example. On the other hand, "a customer" is a variable. It is a kind of proxy that can be replaced by concrete terms if needed. This makes in particular sense for rules, since otherwise we would need a separate rule for each customer. Finally, complex terms are terms that can be computed from other terms. In the example "the turnover of a customer in the year 2002" is a complex term. The *function* "turnover of a customer in a certain time" is an instruction how to build a new term from two other terms. The terms inside a complex term can again be constant, variable or complex.

Terms are *typed*. In line with the object-oriented approach, mandarax types are (arbitrary) java classes. Primitives such as `int` and `boolean` are represented by their respective wrapper classes.

In the last example, terms would be associated with the following types:

```
If the turnover of a customer was more than 100$ in the year 2002 then the customer gets a discount of 5%
        If the turnover of a customer was more than 100$ in the year 2002    // prerequisite
                more than                          // predicate
                the turnover of a customer in the year 2002    // complex term
                        the turnover               // function
                        a customer                 // variable term, type is com.mycompany.Customer
                in the year 2002                   // constant term, type is int
        100 $                                      // constant term, type is float
        the customer gets a discount of 5%         // conclusion
                gets discount                      // predicate
                the customer                       // variable term, type is com.mycompany.Customer
                5 %                                // constant term, type is com.mycompany.Discount
```

Predicates associate terms of a certain type. This is called the structure of a predicate. For instance, the structure of the "gets discount" predicate is `{Customer.class,Discount.class}` (the package names are omitted). This is an array of `java.lang.Class` instances indicating the the predicate is an association between customers and discounts.

With a set of rules like this one can issue queries like `“what is discount customer John Smith qualifies for”`. If there is a fact such as `“the customer John Smith purchased goods for 120 $ in 2002”` then the answer would be `5%`. This statement would be a fact supporting the rule. We then have a rule '`if A then B`', a fact `A` and can prove `B` based on this. This derivation is the responsibility of an inference engine introduced later in this document. The computation performed by an inference engine is not trivial: it performs term replacements (`'a customer'` is replaced by `'John Smith'`), it can handle recursion (it can prove `C` from `A`, '`if A then B`', '`if B then C`') and more. In particular, in mandarax it is not necessary to add facts like `“the customer John Smith purchased goods for 120 $ in 2002”` explicitly, instead these facts can be built on the fly from database queries or java objects.

Creating Logical Objects

Instances of the respective interfaces for rules, facts and terms are created using a *logic factory* object. Mandarax contains a reference implementation for each interface in the `reference` package. The default logic factory can be accessed using

```
org.mandarax.kernel.LogicFactory.getDefaultFactory().
```

By default, this method returns an instance of the implementation class `org.mandarax.reference.DefaultLogicFactory`. The class `LogicFactory` has a “soft” reference to this class (using `Class.forName()`). The reason is that the `kernel` package should not depend on the reference implementation. Therefore, if the reference implementation package is not in the classpath, this method returns `null`.

The methods in the logic factory are simple. For instance, a variable term of the type customer could be created using the following code:

```
LogicFactory factory = LogicFactory.getDefaultFactory();
Term term1 = factory.createVariableTerm(“a customer”,Customer.class);
Discount discount = new Discount(5);
Term term2 = factory.createConstantTerm(discount);
Term[] terms = {terms1,term2};
Class[] predicateStructure = {Customer.class,Discount.class};
Predicate predicate = new SimplePredicate(“gets discount”,predicateStructure);
Fact fact = factory.createFact(predicate,terms);
```

In a situation where `Discount` is an interface and `DiscountImpl` is an implementation class, the type must be specified. The code could look as follows:

```
LogicFactory factory = LogicFactory.getDefaultFactory();
Term term1 = factory.createVariableTerm(“a customer”,Customer.class);
Discount discount = new DiscountImpl(5);
Term term2 = factory.createConstantTerm(discount,Discount.class);
Term[] terms = {terms1,term2};
Class[] predicateStructure = {Customer.class,Discount.class};
Predicate predicate = new SimplePredicate(“gets discount”,predicateStructure);
Fact fact = factory.createFact(predicate,terms);
```

In the `util` package there is a similar class `LogicFactorySupport` adding further utility methods to create facts, terms, rules and queries.

Clause Sets

In logic programming, facts and rules together are often called *clauses*. Most rule systems use a collection or list of clauses to represent knowledge. In real world applications, this approach has some fundamental flaws. Consider the following example:

If the favorite color of a customer is red then send him the special offer ABC.

Most rule systems would represent this rule together with facts like:

The drivers license number of Max is 12345.
The drivers license number of John is 424242.
..

The disadvantage is that these facts are usually built from data stored in some kind of database. The database could be anything from local files to web sources, most often it is a relational database. Therefore we must replicate the database into the knowledge base (which is usually in memory). This can be very expensive if the database base is large. What is more, many applications will need real time facts. This would mean that knowledge base and database must be synchronized at the very time a query is issued.

To avoid this situation, Mandarax uses clause sets instead of clauses. Clause sets are basically iterators over collections of clauses. In the example given, such a clause set could be defined around a JDBC-SQL query: the query returns a result set and the clause set builds facts from the records at *query time*. Replication of data is therefore not necessary any more[2]. Clause sets are downward compatible: rules and facts are considered as singleton clause sets.

Clause sets have many applications. We just sketch a few:

1. Clause sets are used by mandarax in order to deal with rules with prerequisites connected by “or”. Such a rule `“if A or B then C”` is interpreted as a clause set comprising the two simple rules (clauses) `“if A then C”` and `“if B then C”`.
2. Imagine you want to design a “learning” knowledge base. Such a knowledge base could contain a clause set with an iterator that prompts the user to confirm a fact. If confirmed, the clause set contains this one fact; if rejected, it contains no facts at all.
3. Collaboration between agents could be implemented by a clause set that issues a query to another inference engine / knowledge base pair, the answer is the set of clauses defined by the query and the respective variable substitutions in the result set.
4. Clause sets based on an SQL queries. Mandarax contains several classes supporting this, for details see “SQL Predicates and SQL Clause Sets” on page 16.
5. The Mandarax `AutoFacts` class build facts dynamically from sets of objects. For details see AutoFacts on page 19.

Maintaining the Knowledge Base

Adding and Removing Knowledge

[2] Besides the respective implementation has a built-in transparent caching mechanism that can be used for performance improvement if real time data are not required.

Knowledge bases are containers for clause sets. The interface for knowledge bases is `org.mandarax.kernel.KnowledgeBase`. The basic methods defined in this interface are `add()` and `remove()`, both taking a clause set as parameter. The `removeAll()` method clears the knowledge base.

Knowledge bases propagate changes using *clause set change events*. The respective methods to register and unregistered clause set listeners are also declared in `KnowledgeBase`. However, not all changes in the fact base will trigger clause set change events to be fired. For example, if the clause set describes a set of facts built on the fly from the result set of an SQL query, the fact base changes whenever the database changes. Usually, the knowledge base will not be aware of this and won't fire an event. But such changes affect only clauses, not the clause sets. And the inference engine pulls knowledge from the knowledge base when processing a query and therefore does not rely on the change events fired by the knowledge base.

Retrieving Knowledge

The content of a knowledge base can be easily retrieved using `getClauseSet().` Clause sets are indexed by a *key*. Clause sets implement a `getKey()` method to provide this key. Although this is not enforced (`getKey()` returns in fact an instance of `Object`), the key is usually a predicate (the predicate of the fact or, in case of rules, the predicate of the head of the rule). Knowledge bases may index clause sets by this key in order to ensure fast access to clause sets by inference engines. One flavor of `getClauseSets()` takes a key object as parameter.

Ordering Knowledge

The interface `org.mandarax.kernel.ExtendedKnowledgeBase` extends `KnowledgeBase` and provides additional methods to move clause sets up, down, to the button and to the top. This implies some linear structure in the knowledge base that can be seen as the order imposed by *priorities* associated with clause sets. The lists returned by `getClauseSet()` should respect this order. This has certain implications for the inference engine: the engine will try to use the first clause sets first in proofs. Hence different orders might lead to different query results.

In version 2.2, a comparator property has been introduced (in `ExtendedKnowledgeBase`). If a comparator is set, knowledge objects will be ordered by this comparator. The package `org.mandarax.util.comparators` contains a default comparator class `DefaultClauseSetComparator`. This class implements some strategies like:

- Prefer facts and SQL clause sets over rules
- Prefer facts with less variables
- Prefer rules with more prerequisites
- Prefer rules with less negated prerequisites

These strategies basically express that knowledge we consider to be more precise or more sound should be preferred over vague knowledge.

Issuing Queries

Understanding the Inference Process

The main reason that we set up and maintain a knowledge base is to retrieve knowledge from it. For this purpose, a *query* must be issued A query is more or less a fact that contains some (query) variables. In the answer of the query we expect that the variables are replaced by objects.

Example

Q: How much discount gets customer X?
A: X gets 5% discount. (The formal answer is X / 5%)

Q: Who killed Mr. Holmes and where did it happen?
A: The hound of Baskerville killed Mr. Holmes in the swamps. (The formal answer is`who/hound of Baskerville`; `where/swamps`).

The algorithm used to find these replacements is implemented in a so-called *inference engine*. The interface for inference engines is `org.mandarax.kernel.InferenceEngine`. The inference engine takes a query[3] and a knowledge base as input and returns an instance of `org.mandarax.kernel.ResultSet`. This result set instance knows the replacements and has an object representation of the proof supporting the result.

Backward vs. Forward Reasoning

The mandarax inference engine uses *backward reasoning*, and the reference implementation uses an object-oriented version of backward reasoning similar to the algorithm used in Prolog. On the other hand, most commercial rule systems such as ILOG and popular open source solutions like CLIPS and JESS use forward reasoning, in particular an algorithm called RETE. This algorithm keeps the derivation structure in memory and propagates changes in the rule and fact base. This can be very effective. However, we see the following advantages in the backward reasoning approach:

1. The effective integration of facts from arbitrary data sources. In forward reasoning, additional software must watch these data sources and must propagate changes in those data sources in order to keep the derivation structure up to date. This can be difficult in particular for relational databases: databases do not propagate changes in their data[4], and if real time facts are needed the database and the fact base must be synchronized before a query is issued!
2. This is part of a more general pattern: most business solutions implement a *pull model*, like a sales transaction initiated by a web store customer. In particular, system architectures using a web clients and SQL databases are usually based on a pull model as they are using request based protocols such as HTTP and SQL network protocols (SQLNet etc). In this system landscape it is (at least) very difficult to implement a push model. An example would be a database trigger that triggers an event that is then propagated through the middle ware layer to the web client where a web page is updated automatically without user interaction[5]. The query driven backward reasoning fits

[3] Note that in versions prior to 1.9 there was no separate query type. Instead, facts were used. A query is more general: queries can contain many query facts. To facilitate the migration of applications from 1.8 to 1.9, **`FactImpl`** implements the **`Query`** interface and can therefore be casted to **`Query`**.
[4] Some databases have an event concept called triggers. But connecting triggers with java clients is not straightforward and there are no standards supporting this.
[5]There is no doubt that this can be done in principle, for instance with triggers sending JMS messages to the middle ware and with a web server using pushlets to update the web site. The claim we are making is that HTTP and SQL network protocols have not been designed for this and that the quality of such a solution is inherently poor.

perfectly into this architecture. A push model architecture can still be implemented using mandarax, for instance by using a demon (kind of push-pull adapter) that frequently issues queries and fires events (pushes information) depending on the result of the query. The mandarax ECA subproject uses this approach.

3. Many rule systems are working on large fact bases (like the customer database) but a rather small rule base (by small we mean some hundreds, but not ten thousands of rules). Large rule bases quickly become extremely difficult to understand and to maintain. For such small rule bases, there is no significant difference regarding performance between backward and forward reasoning. On the other hand, mandarax takes advantage of existing technologies managing large fact bases (like SQL databases with services such as indexing and query optimizing).

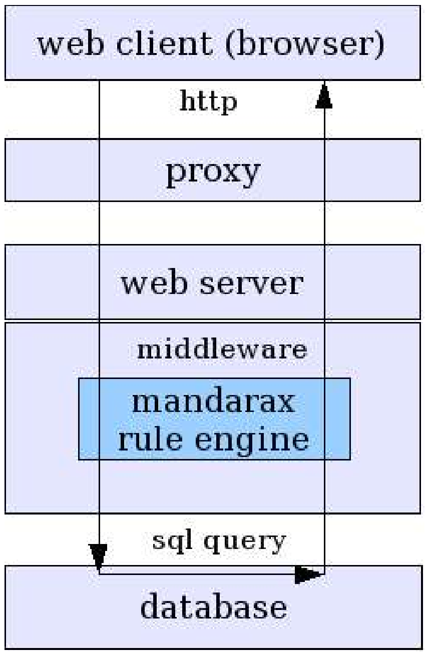


Figure 1 A query based rule engine in a typical 3-tier (pull) architecture

Using the Semantics

Backward reasoning algorithms are based o formal (syntactical) reasoning. They take only the syntax of the terms and other logical objects into account. Suppose we have a rule "if 2+2=4 then math works" and a query "does math work?". The pure algorithm could not proof this since "2+2=4" is just a sequence of objects and the algorithm does not know how to interpret + and =. Of course, most rule system are aware of this and implement some kind of support for these situations. This means, they integrate the meaning or the *semantics* of the respective expressions. Mandarax goes one step further and distinguish between semantic and non-semantic objects. There is an interface `org.mandarax.kernel.SemanticsSupport` with a single method **`boolean`** `isExecutable()`. This method can be used to find out whether the respective object supports the semantics or not. Terms, rules, facts, predicates and functions all implement this interface.

We illustrate how this works with an example:

Example

Imaging there is a rule "if the turnover of a customer X is more than 100 give him a discount of 5%" and a query "get the discount of customer Tom". The turnover of a customer X function is defined by a SQL query to a relational database, something like `"SELECT SUM(TURNOVER) FROM CUSTOMER_TRANSACTIONS WHERE NAME=?".` How to do it in detail is explained on page 17. The inference engine can answer the query without any additional fact using the meaning of "is more than" (this is actually the meaning of the < operator in java) plus the result of performing the SQL query with the input parameter "?:=Tom".

This process is completely transparent: after every proof step the inference engine tries to simplify the list of goals using this semantic evaluation. In order to ease the composition of semantic objects (predicates and functions), the `org.mandarax.lib` packages contain numerous predicates and functions wrapping java functionality for numbers, strings and dates. The `org.mandarax.sql` package contains generic predicates and functions using the semantics of database queries, and the `org.mandarax.kernel.meta` package contains generic classes to integrate the semantics of java methods.

Working with Results

The inference engine returns the result as an instance of `org.mandarax.kernel.ResultSet`, a structure very similar to the result set used in JDBC[6]. The methods `next()` and `previous()` can be used to navigate through the result set. Both methods return `true` if the cursor has been moved to a valid position and `false` otherwise. Note that the number of results does not only depend on the query and the knowledge base, but also on the cardinality constraint passed to the query. Only if the constraint is `InferenceEngine.ALL` all results will be returned. The specification leaves it open whether all results should be fetched at once or 'on demand'. In fact, the later 'lazy' approach is more suitable for (enterprise) server applications. However, the current implementation of an inference engine supporting multiple results (`ResolutionInferenceEngine2`) fetches all results at once.

If the result set cursor is positioned on a valid position, results can be fetched similar to fetching column values in a JDBC query. Instead of a column name, the variable (or the type and the value of the variable) must be passed. The following code shows how to issue a query and how to fetch values from the result set:

[6] Prior to Mandarax version 1.7 there was a different query API using a `Result` class. The reason to move to a more JDBC like API is to support JDBC experienced programmers in using mandarax.

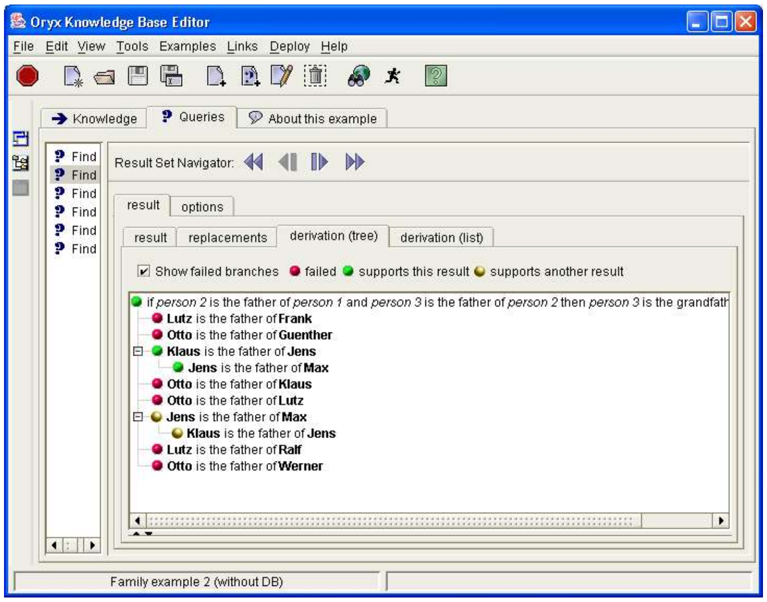
Example

```
LogicFactorySupport lfs = new LogicFactorySupport();
KnowledgeBase kb = ..;
Predicate isOncleOf = ..;
Fact queryFact = lfs.fact(isOncleOf,lfs.variable(“onlce”,Person.class),lfs(new
Person(“Max”)));
Query query = lfs.query(queryFact,”get the oncle of Max”);
InferenceEngine ie = new ResolutionInferenceEngine2();
ResultSet rs = ie.query(query,kb,ie.ALL,ie.BUBBLE_EXCEPTIONS);
while (rs.next()) {
        System.out.print (“The oncle of Max is ”);
        System.out.println(rs.getResult(Person.class,”oncle”));
}
```

Note that some of the methods used may throw exceptions. Exception handling code is omitted in the last example. The internal exception handing of the inference engine can be configured using the exception handling policy parameter passed to `query()`. If the parameter is set to `BUBBLE_EXCEPTIONS`, each exception encountered when resolving clause sets leads to an exception thrown by the inference engine and the inference process is canceled (pessimistic approach). If the policy is set to `TRY_NEXT,` the inference engine ignores exceptions thrown by clause sets and tries to continue the derivation with other clause sets (optimistic approach).

Although there is no separate result set meta data object as in JDBC, a list of variables replaced by the query can be obtained by `getQueryVariables()`.

Another important issue is to know *how* the inference engine computed these results. For this purpose, `ResultSet` has the `getProof()` method that returns the *derivation*. The derivation is organized as a tree of *derivation nodes*. Inference engines return a tree comprising all nodes supporting the derivation. Optionally, nodes representing clauses not supporting the derivation can be part of this tree as well but should respond to `isFailed()` with `true`. The derivation can be easily visualized, for instance by using a swing `JTree` component. Each node knows the clause that has been applied and the unification used. This should help the user to understand the derivation and to support the *rule base development life cycle*: run a query, understand the effects of rules and make the rule base better. Another helpful tool that can be used in order to analyze the derivation is the class `org.mandarax.util.ProofAnalyzer`.



✓ Show failed branches ● failed ● supports this result ● supports another result

```
● if person 2 is the father of person 1 and person 3 is the father of person 2 then person 3 is the grandfath
    ● Lutz is the father of Frank
    ● Otto is the father of Guenther
  ⊟ ● Klaus is the father of Jens
        ● Jens is the father of Max
    ● Otto is the father of Klaus
    ● Otto is the father of Lutz
  ⊟ ● Jens is the father of Max
        ● Klaus is the father of Jens
    ● Lutz is the father of Ralf
    ● Otto is the father of Werner
```

Figure 2 Visualization of a derivation with a swing tree component (using the Oryx extension)

Result Set Filters

Result set filters can be used in order to manipulate the result set. Filters themselves implement the `ResultSet` interface and are usually applied as wrappers (wrapping another result set). In particular, filters can be used to achieve the following tasks:

– sort results in a result set
– apply aggregations functions
– filter results using boolean conditions (e.g., prefer results with short simple derivations)
– identify results (e.g., identify results with the same set of replacements but different derivations)
– any combinations of the above mentioned filters (using a filter pipe).

Some filters are implemented in the `org.mandarax.util.resultsetfilters` package. In particular, the following implementing classes are in this package:

Class name	*Description*
`OrderByFilter`	Sorts results.
`GroupByFilter`	Aggregated result.
`WhereFilter`	Filters results using a boolean expression.

Table 2 Mandarax predefined result set filters

As the names indicate, these filters provide functionality similar to the respective features in relational databases. The following example show how to use an `OrderBy` filter:

```
ResultSet rs = ... // result set returned by inference engine
// define ORDER BY conditions
VariableTerm NAME = (VariableTerm)lfs.variable("name",String.class);
VariableTerm FNAME = (VariableTerm)lfs.variable("fname",String.class);
OrderByCondition[] cond = new OrderByCondition[]{new ASC(NAME),new ASC(FNAME)};
// sort results by (the replacements for) name and first name
rs = new OrderByFilter(rs,orderByConditions);
```

Customizing the Inference Engine

The reference implementation package contains five inference engine implementation: `ResolutionInferenceEngine`, `ResolutionInferenceEngine2`, `Resolution InferenceEngine3` , `ResolutionInferenceEngine4` and `DefaultInferenceEngine`. All **`Resolution`**`InferenceEngine?` classes extend a common super class `AbstractResolutionInferenceEngine`. This super class can be used as a base class for alternative implementations. The main difference between the implementations is the set of features they implement.

Engine	*Multiple results*	*Negation as failure*	*Cut*
`ResolutionInferenceEngine`	NO	NO	NO
`ResolutionInferenceEngine1`	YES	NO	NO
`ResolutionInferenceEngine2`	YES	YES	NO
`ResolutionInferenceEngine3`	YES	YES	YES

Table 3 `InferenceEngine` **Implementations**

Our policy is to keep older implementations alive and to freeze development at a certain stage. The advantage for ongoing projects is to have a more reliable engine not affected by the risks of introducing new features. On the other hand, these older engines are in general slightly faster. The supported features can be discovered at runtime using the `getFeatureDescriptions()` method.

`DefaultInferenceEngine` is simply a wrapper class that uses the latest stable implementation as a delegate.

Engines are highly customizable. In particular, the following “sub algorithms” can be customized:

1. The selection policy used to select the next goal to proof (the default is `LeftMostSelectionPolicy`)
2. The unification algorithm (the default is `RobinsonsUnificationAlgorithm`)
3. The loop checking algorithm (the default is `NullLoopCheckingAlgorithm` – that means no loop checking at all)

In many cases these defaults work fine and modifications are not necessary. However, if the rule base has circular dependencies and the proofs are running into loops, it is useful to replace the loop-checking algorithm by `org.mandarax.reference.DefaultLoopChecking Algorithm`. This algorithm is neither a correct nor a complete loop checker in the strict mathematical sense, but very effective to detect loops practical applications will encounter (kind of 80/20 approach). This loop checker is suitable if one can estimate the recurrence patterns of applied clauses causing an infinite loop (number of recurrences indicating that we are in a loop, and the max number of recurring patterns).

The reference implementation classes both have a property `maxsteps`. This integer value specifies the maximum number of derivation steps the inference engine should perform before it gives up. This is another mean to prevent looping.

Note that besides the sub algorithms mentioned and the `MAXSTEPS` variable inference engines do not have state information. In particular, there is no query related information kept in instance variables. Therefore, multiple threads can easily share a single inference engine. Only if different engine configurations are required multiple instances (one per configuration) are necessary. On the other hand, the knowledge base implementations have synchronized access methods. This might not be appropriate for all application types. But considering that mandarax is open source it is always an option to copy the class and remove the synchronize constraint from the access methods.

The negation as failure feature can be used in order to negate prerequisites in rules. The meaning of a rule `'if not A, B then C'` is `'if A can not be derived and B can be derived then C can be derived as well'`. That means that the proof of `A` must fail to apply this rule successfully. Therefore, query processing with rule bases containing negation as failure can be very expensive.

Integrating Knowledge from Databases

Connecting to the Database

When integrating knowledge from relational databases, we must first establish connections to the database. Mandarax uses data sources instead of raw connections and therefore depends on the `javax.sql` standard extension package.

SQL Predicates and SQL Clause Sets

SQL predicates are basically database tables (or views) or a vertical part (i.e., selected columns) of a table or view. An SQL predicate consists of the following components:

1. A data source that contains the info how to connect to the respective database. The data source is an instance of `javax.sql.DataSource`, therefore the `org.mandarax.sql` package depends on this extension package.
2. A name.
3. A query. This query usually does not have a `WHERE` clause since the number of rows does not matter. The purpose of the query is only to describe the structure (the columns with their data types) of a result set. E.g., `SELECT * FROM STUDENTS`.
4. A type mapping. This type mapping describes how to map the values in the result set to java types. The type mapping is an instance of `org.mandarax.sql.TypeMapping`. The class `DefaultTypeMapping` implements this interface and provides a type mapping that should be sufficient for most purposes. By default, SQL predicates use this type mapping. If the type mapping is `null`, the SQL predicate tries to issue a query in order to

get the type mapping from the JDBC driver (using the method `getColumnClassName()` in `ResultSetMetaData`). Warning: not all JDBC drivers[7] support this feature!

SQL clauses are implemented by `org.mandarax.sql.SQLPredicate`, and access methods for the respective features are implemented there.

An sql clause set (class `org.mandarax.sql.SQLClauseSet`) is more or less an SQL predicate plus an optional `WHERE` clause restricting the rows (and therefore the facts) in the result set. In addition, SQL clause sets have an optional caching mechanism that can be used in order to re-use facts built from the query. The mechanism can be configured using `setCacheTimeout()`, expecting a number of milliseconds as parameter. In order to switch the cache off, pass `SQLClauseSet.NO_CACHE` as parameter. This is the default setting.

Example

A predicate that associates father and son could be defined as follows:

TABLE FAMILY

NAME	**FATHER**
Max	Jens
Jens	Klaus
Klaus	Otto
Guenther	Otto
Otto	null

```
DataSource dataSource = ..;
Class[] struct = {String.class,String.class};
SQLPredicate predicate = new SQLPredicate();
predicate.setName("is father of");
predicate.setQuery(“SELECT NAME,FATHER FROM FAMILY”);
predicate.setDataSource(dataSource);
predicate.setStructure(struct);
```

SQL clause sets and SQL functions (see next section) both have a boolean property `closeConnection`. If this property is set to `true`, the connection used will be closed after each database access. This can be useful in an application server environment where the data source serves (wrapped) connections from a connection pool, and closing a connection returns the connection to the pool of available connections.

SQL Functions

SQL functions are functions based on an SQL query. The idea is to take certain input parameters (usually in the `WHERE` clause), and to build an object from the one row returned. It is the nature of a function that it returns exactly one value, therefore an SQL function will throw an exception when it is invoked and there are either no rows or more than one row in the (JDBC) result set[8]. SQL functions are implemented by `org.mandarax.sql.SQLFunction.` SQL functions have the following properties:

[7] In particular, we have encountered problems with the MySQL driver **`org.gjt.mm.mysql.Driver`**.

[8] This is similar to the usage of SQL queries as functions in the PL/SQL **`SELECT INTO`** statement.

1. A data source (see SQL predicates)
2. A name.
3. A query string. This query string contains one or many ?s as placeholders for variables. Note that internally the query string is translated into a prepared statement.

An object relational mapping (instance of `org.mandarax.sql.ObjectRelationalMapping`). This mapping is responsible to take a record (the one row in the result set) and convert it into an object. The most common case seems to be that the result set contains only one value (see example), in this case the implementation class `OneColumnMapping` can be used.

Example

A function that takes a customer name and returns the number of transactions of this customer could be defined as follows:

```
String[] struct = {String.class};
DataSource dataSource = ..;
SQLFunction function = new SQLFunction();
function.setDataSource(dataSource);
function.setQuery(“SELECT COUNT(*) FROM CUSTOMER_TRANSACTIONS WHERE CUSTOMER=?”);
function.setObjectRelationalMapping(new OneColumnMapping(Integer.class));
function.setName("number of all transactions of a customer");
function.setStructure(struct);
```

TABLE CUSTOMER_TRANSACTIONS

ID	CUSTOMER	AMOUNT
1	Jim	100.00
2	John	1200.00
3	Tom	49.99
4	Jim	99.95
5	Tom	42.00

Integrating the Java Object Model

JFunctions and DynaBeanFunctions

Java methods can be regarded as functions in the mandarax sense: they take objects (= terms) as input and return another object. The object that receives the method can just be considered as another parameter. For instance, when sending `indexOf(txt)` to `aString`, the associated function has the two parameters `{aString,txt}` and returns an `Integer` instance.

This simple principle is supported by the Mandarax class `org.mandarax.kernel.meta.JFunction`. A `JFunction` wraps a java method. A `JFunction` is executable – is all parameters are concrete objects (constants and not variables or complex terms), the associated method is invoked and the result returned. For instance, if we had a complex term `indexOf(“abc”,”a”)` (with `indexOf` being the function wrapping the `String.indexOf()` method), the term could be simplified to the constant term **`“0”`** using reflection.

In version 2.2 a related implementation named `DynaBeanFunction` has been added. Such a function is useful in order to integrate accessor methods in map like objects. Instead of having one method per property there are generic methods that take the property name as parameters. Often the methods are stored internally in a map and the property name is used as the key. A `DynaBeanFunction` is defined by a method (with one string type parameter) and a string (the property name).

JPredicates

In a similar manner, methods returning a boolean value can be considered as predicates. Such a method associates the object that receives the method and the parameter. For instance, consider the method `equals()` defined in `Object`. An expression `obj1.equals(obj2)` is interpreted as a fact obj1=obj2. If both objects are known, the fact can easily be verified by invoking `equals` on `obj1` with the parameters `{obj2}` and check whether the result is `true`.

Example

Consider the following situation:

- **`Person`** is a class with a property `name` (equipped with the associated set- / get methods)
- = is the predicate wrapping `Object.equals()`
- `getName()` is the function wrapping `Person.getName()`
- Consider the following rule: if x.getName()=”George Bush” then x is very important

When issuing a query like “Is object <George Bush> very important”, the inference engine can replace “x” by (the instance of Person!) “<George Bush>”, apply the `getName()` method that yields (the string!) “George Bush”, and invoke equals. This yields `true`, therefore the prerequisite has been proved and the answer is yes (he is important indeed).

AutoFacts

AutoFacts are generic clause sets. AutoFacts (class `org.mandarax.util.AutoFacts`) facilitate the task of integrating facts generated from data into the knowledge base at query time . The basic idea of auto facts is to take one or many predicates and a map associating types (java classes) with instances (collections). AutoFacts will then investigate all combinations of those instances, apply the predicate(s) and return a clause if this yields true. Therefore the predicate must be executable (see Using the Semantics on page 11).

Example

Assume we have an instance af of a subclass of **`AutoFacts`** with **`getExtension`** returning (the **`Integer`** instances) 1,2 and 3 for **`Integer.class`**. Now assume that we perform **`af.clauses(1<x,null)`**, where **`1<x`** stands for the fact consisting of the **`IntArithmetic.LESS_THAN`** predicate, a constant term wrapping the integer 1, and a variable term of the type **`Integer`** named "x". Then the iterator returned iterates over two facts: “1<2” and “1<3”. If we perform **`af.clauses(x<y,null)`**(two variables!), the iterator iterates over three facts “1<2”, “1<3”, “2<3”. This will also work if the parameter(s) contains functions (with a known semantics), e.g., the iterator returned by **`af.clauses(x*x<x+x,null)`** (where * and + stand for
the respective function defined as static members in **`IntArithmetic`**) will iterate over one fact only: 1*1<1+1.

If **`clauses()`** is used without parameters, the set of facts iterated is defined as follows:

- The predicate of each fact must be in the array returned by `getPredicates()`.
- For each type of any slot of the predicate, the extension is computed, and a fact is built for each combination of these extensions. The predicate is then executed and the fact is only returned if this yields true.

Example

If `getPredicates()` returns **{**`IntArithmetic.LESS_THAN`**,**`IntArithmetic.EQUALS`**},** and the extension for `Integer.class` is defined as {1,2,3}, then `clauses()` returns an iterator iterating over the following facts: "1=1" ,"2=2" , "3=3" , "1<2", "1<3" and "2<3".

The Mandarax Libraries

The mandarax lib packages contain a rich set of predefined functions and predicate. This includes standard functionality for integer and double arithmetic and for the manipulation of text and dates. These predicates and functions can be accessed as static members of the following classes:

- `org.mandarax.lib.math.IntArithmetic`
- `org.mandarax.lib.math.DoubleArithmetic`
- `org.mandarax.lib.text.StringArithmetic`
- `org.mandarax.lib.date.DateArithmetic`

In addition, there is a cut predicate that implements the cut functionality used in logic programming. Cut should be used with caution only be people who understand logic programming – cut prunes the derivation tree and can therefore have several side effects. Furthermore, cut is not supported by all inference engine implementation. In particular, `ResolutionInferenceEngine4` supports cut.

The XKB XML Interface

Managers and Drivers

The XKB package(s) provide support to serialize knowledge bases using XML. The XKB packages use the JDOM library that is not part of the JDK distribution[9], but can be obtained for free from www.jdom.org.

In version 3.0 development for XKB has stopped. Support for newly introduced features (such as named slots) will not be added to XKB. The ZKB module should be used instead.

[9] Although JDOM has been accepted by the Java Community Process (JCP) as a Java Specification Request (JSR-102).

The basic functionality of the interface is described by the interface `org.mandarax.xkb.Driver`. The driver has an `importKnowledgeBase()` method that expects a (jdom) document parameter and returns a `KnowledgeBase`, and an `exportKnowledgeBase()` method that works the other way around. Both methods may throw an `XKBException`. In addition to these methods, a driver supports to a certain extend introspection: there are various methods returning boolean indicating whether the driver supports a certain feature of not. The driver also declares a `getDTD()` method returning a string indicating where the DTD of the supported XML format is defined.

The driver is usually wrapped by an `XKBManager`. The manager has several convenience methods, e.g. to read (write) a knowledge base from (to) a stream, a reader/writer, a URL or file. The XKB manager also allows users to plug-in their own components used to parse XML.

The RuleML Driver(s)

The `org.mandarax.xkb.ruleml` package contains drivers for the XML format defined by the RuleML group. Currently only the latest versions RuleML 0.8 and RuleML 0.8.1 are supported, the implementing classes are `RuleML0`**`_8Driver`** and `RuleML0`**`_8_1Driver`**. Note that this format is minimal, and a lot of features cannot be mapped into this format. In particular, RuleML supports neither functions nor types! The author works with the RuleML group and hopes that later versions will have more expressive power. The difference between both RuleML drivers is that the 0.8.1 driver supports queries.

The Generic Driver and How to setup your own Driver.

The `org.mandarax.xkb.framework` package contains a little framework that can be used in order to produce drivers very easily. The key idea is that there is a class `GenericDriver` that forwards most of the work to `XMLAdapters`. The adapters know how to convert an object to a (jdom) element and vice versa. The names of the respective methods are `exportObject` and `importObject`. We pass a reference to the generic driver with this methods so that the adapters can delegate the export / import of certain "sub objects" back to the driver using one of the adapter finder methods in `GenericDriver`. The driver will then locate the appropriate adapter in its registry and the adapter can delegate the task to this adapter. In order to locate adapters, adapters publish a symbolic name (usually a constant like **`GenericDriver.RULE`** for adapters importing/exporting rules) and the name of the associated XML tag.

In order to facilitate the implementation of adapters we provide two abstract super classes, `AbstractXMLAdapter` and `CachedXMLAdapter`. The first class implements some "shortcuts" for exporting / importing children objects/elements. The purpose of the second class is to identify certain objects. For instance, consider the case that we have three facts sharing the same predicate. Using a subclass of cached adapter we can make sure that there is only one predicate if we read the facts from an XML file. This makes sense: e.g. editing the name of the shared predicate in a graphical user interface should apply to all facts. This is achieved by storing an additional unique object id attribute. The identification of objects happens in a map. By default, maps identify keys using **`equals()`**. But the map interface does not enforce this, there might be implementations identifying objects using ==[10]. A cache instance is passed to the first adapter, if one needs a special cache – this is the place to implement it.

[10] Like the "Identity Dictionaries" in Smalltalk.

The generic driver does not (yet) publish the DTD of the associated XML format. We are currently working on a modular DTD definition where each adapter publishes his part of the DTD.

There is one adapter for objects. Since constants are more or less wrappers around objects, we have to address the issue of general-purpose java serialization. The default bean serializer (`XMLAdapter4Objects`) uses bean introspection to serialize any kind of objects. However, there are limitations. For instance circular objects graphs are not supported. If a certain property of an object must not be serialized, one can use a `BeanInfo` object to describe the serializable aspects of the respective object. We are aware of the serialization support in JDK 1.4 and will integrate this functionality one JDK 1.4 is the established standard[11].

The following example shoes how to set up a driver using the GenericDriver framework.

Example

```
import org.mandarax.xkb.framework.*;
public class MyDriver extends GenericDriver {

// Constructor.
public XKBDriver_1_0() {
      super();
      initialize();
}
// Initialize the object.
// Use some default adapters and a special adapter for facts.
private void initialize() {
      install(new XMLAdapter4ComplexTerms());
      install(new XMLAdapter4ConstantTerms());
      install(new MySpecialAdapter4Facts());
..    …
}}
```

XKB versus RuleML

The XKB1.0/ XKB1.1 driver is not compatible with the current RuleML standard (RuleML 0.8 and 0.8.1, respectively). The reason is that RuleML currently only supports very few of the features implemented by Mandarax. In particular, this applies to the following features:

1. Typing
2. Complex terms and functions
3. Clause sets (only single facts are possible)
4. Integration of SQL data sources
5. Rule bodies connected by OR.

However, we have tried to design the top-level elements in a way that makes them compatible with RuleML 0.8. This will at least facilitate the task of migrating sources between the two formats, and to implement drivers for future RuleML versions.
This top-level compatibility includes the following aspects:

[11] Yet another possibility is the classical object serialization and the embedding of the (binary) data, see java world tip 117 (http://www.javaworld.com/javaworld/javatips/jw-javatip117.html?tip) for details on how to embed binary data in an xml document.

1. The `<imp>` , `<_head>` and `<_body>` tags are used for rules.
2. As in RuleML, the `<and>` tag is used inside the `<_body>` tag, but alternatively, the `<or>` tag can be used as well. While the `<and>` tag is optional in RuleML, it is mandatory in XKB1.0, even if there is only one prerequisite.
3. The `<atom>` tag is used for facts (and prerequisites and conclusions in rules).
4. The `<_opr>` tag is used for predicates. However, RuleML has a subtag `<rel>` inside `<_opr>`. XKB1.0 has other tags indicating the type of predicate (`JPredicate`, `SQLPredicate` or `SimplePredicate`) and the respective properties.
5. The `<ind>` tag is used for constants. However, while the RuleML tag is flat, the XKB1.0 tag has children indicating the type and the value of the constants. The value tag represents a serialized java object or primitive and is therefore a "deep" tag itself.
6. The `<var>` tag is used for variables. However, while the RuleML tag is flat, the XKB1.0 has children indicating the type of the variable. The name of the variable is stored as attribute.

The ZKB Framework: An Alternative XML interface

Overview

In Mandarax version 2.2, an alternative persistence interface called ZKB has been introduced. The Z in ZKB refers to the use of ZIP compression technology. The main reason to introduce ZKB was that the XKB framework is responsible for general-purpose object serialization. While ZKB still uses XML to represent knowledge bases and items like facts, rules and terms, the serialization of references objects is delegated to a separate object called Object Persistence Service (OPS).

The purpose of an OPS is to generate names for objects, the interface is similar to a JNDI context. The XML document representing the knowledge base uses these generated names to represent objects, and the OPS binds objects to these names. Furthermore, the OPS can export the bindings to a stream, and import them from a stream. Finally, the OPS has a lookup method that can be used in order to lookup objects by name. Mandarax contains two OPS implementations: one based on binary serialization, and one based on XML object serialization[12] .

This separation results in two files that are needed to represent the knowledge base: the RuleML like XML file representing the knowledge base, and a (binary or XML) "re source" file containing the serialized objects references by the knowledge base. The ZKB manager simply combines these two files into one using zip compression. A third file containing meta information (e.g. about the OPS used, version info etc) is part of the zip file as well. Note that a similar approach is used in some office packages.

The ZKB Manager

Application programs interact with ZKB via the `ZKBManager`. This class defines simple methods to export / import a knowledge base from / to a file. Moreover, an additional object called *attachment* can be saved together with the knowledge base. As this object can be a container, this means that *many* associated objects can be made persistent together with the knowledge base.

[12] This requires JDK 1.4 or better.

The following code listing shows how to set the OPS, and how to export / import a knowledge base.

```
ZKBManager zkbMgr = new ZKBManager ();
// configure zkb manager
// next line is optional – specifies xml representation of the kb
zkbMgr.setDriver (new ZKBDriver_1_0());
// set OPS – use binary serialization
zkbMgr.setOps(new BinarySerializationOPS());
// get kb from somewhere
KnowledgeBase kb = … ;
// where to save (using a more general stream is also possible)
File f = new File("kb.zkb");
// export – 1 line !
zkbMgr.exportKnowledgeBase(f,original);
// import – 1 line !
kb = zkbMgr.importKnowledgeBase (f);
```

ZKB versus XKB

In ZKB the task of serializing objects is completely outsourced to standard java technology. This makes it the better choice in particular if the object model references by the knowledge base is complex (e.g., if it has circular references) . However, referenced objects may have to satisfy certain prerequisites to be handled correctly by the OPS like implementing the `Serializable` interface (binary serialization OPS). The XML serialization OPS only works with JDK 1.4 or better. At least binary serialization is not a suitable technology for long-term persistence of objects – changes in the class definitions might make the respective files useless.

Implementing a Custom OPS

There are situations when it makes sense to implement a custom OPS. A typical case would be making data sources (references in SQL clause sets) persistent. In many enterprise solutions, such data sources are obtained using JNDI. Instead of serializing the data source it would be more useful to implement an OPS that associates the data source with its JNDI name.

The Mandarax JDBC Driver

The Driver Design

The Mandarax JDBC driver provides an interface applications can use to connect to a mandarax knowledge base as it was a (relational) database. Access to the knowledge base is read only. While it is rather easy to map SQL queries to mandarax queries there is no natural SQL counter part for mandarax functionality such as inserting / updating rules.

The JDBC driver translates SQL queries into mandarax queries, computes mandarax results sets and converts them back into SQL result sets. This translation is based on the following assumptions:

1. Mandarax predicates are considered as “tables”. Therefore, predicate names (at least for predicates used in queries) must be unique[13]. This is not enforced by mandarax.
2. In the database meta info, predicates are tagged with a special table type 'PREDICATE' .
3. `SQLPredicate` term slots are considered as “columns”. New in mandarax 3.0 is the possibility to name slots using the `setSlotNames()` method in predicate. By default, slot names are generated and the names used are slot1, slot2 etc. Note that only the newer ZKB persistently mechanism will save slot names correctly.
4. The mandarax adds columns automatically to the result set. Currently, there is one *pseudo column*, the values in this column represent the derivation that has been used in order to compute the respective result.
5. The mandarax JDBC driver supports only one table per query, in particular joins are not supported. Support for joins will be added later, there is a natural translation from joins to mandarax queries with multiple query facts.

Obtaining a Local Connection

A local connection is a connection within the current JVM instance. A local connection can be obtained as follows:

```
import java.sql.*;
..
// load driver
Class.forName("org.mandarax.jdbc.DriverImpl");
// get connection
Connection con = DriverManager.getConnection("jdbc:mandarax:zkb:kb.zkb");
```

Mandarax URLs start with `'jdbc:mandarax'` followed by a *sub protocol handler* indicating how to load the knowledge base and a parameter indicating the location of the knowledge base. The following table shows the currently supported sub protocol handler.

Driver URL pattern	*Description*
`jdbc:mandarax:zkb:<file_or_url>`	The knowledge base is loaded from the given file or url as ZKB knowledge source.
`jdbc:mandarax:xkb:<file_or_url>`	The knowledge base is loaded from the given file or url as XKB knowledge source.
`jdbc:mandarax:ruleml:<file_or_url>`	The knowledge base is loaded from the given file or url as RuleML knowledge source.
`jdbc:mandarax:ref:<factory_class:param>`	The driver tries to instantiate the class with the given name, tries to cast it to `KnowledgeBaseFactory` and calls the method `getKnowledgeBase(String param).`

Table 4 Mandarax JDBC sub protocols

Obtaining a Network Connection

[13]There are various polymorphic predicates in the lib packages. These predicates are usually not used as query predicates.

A network connection is a connection to a mandarax server located on a different (server) computer. The network protocol used in `http`. On the server side, mandarax is deployed as servlet. The ANT build script has a target named JDBC. This target creates a war file. This war file can be easily deployed by copying it into the `webapps` folder of the web server (e.g., Tomcat 4).

The network protocol used can be replaced easily and in fact mandarax contains a second protocol implementation: a protocol that "fakes" a network. This is mainly used for testing the serialization mechanism, This mechanism is based on JDK 1.4 XML object serialization. The URL syntax for network URLs is as follows:

```
jdbc:mandarax:net:<local_subprotocol>:</exampledata/example-family.zkb>@<server-url>
```

For instance, if the war file built by ant is deployed on a local tomcat installation, the zkb family knowledge base can be access using the following URL:

```
jdbc:mandarax:net:zkb:/exampledata/example-family.zkb@http://localhost:8080/mandarax-
server/jdbcserver
```

The server can be tested using a browser and pointing it to the following URL:

```
http://localhost:8080/mandarax-server
```

To use the local (fake network) protocol implementation, use URLs with the `net` token (as opposed to local URLs) but without the `@<server-url>` token.

A remark on firewalls and proxies. The network driver has been design to work with proxies. In order to support proxies, the proxy details (address, user information) must be passed to the client JVM as system properties. See the java documentation for details.

The JDBC build target builds a directory `build/jdbc/client` that contains all libraries needed by the client.

Issuing Queries

A query can be issued using either a statement or a prepared statement. Once the driver is loaded and the connection has been established, the mandarax knowledge base can be queried like a real relational database such as Oracle or MySQL. In many situations using prepared statements is recommended for two reasons: the SQL statement is parsed only once (this yields advantages in terms of performance) and there is a simple and convenient API to 'inject' real objects (i.e., objects that are neither strings, numbers nor dates and therefore difficult to represent as literals) into the SQL statement. The following SQL syntax features are currently supported by the driver:

1. `SELECT *`
2. `SELECT <column list>`
3. `SELECT DISTINCT`
4. `SELECT COUNT(*).`
5. `WHERE` clauses with simple conditions containing the following operators: `=, !=, <, > , <= , >=`.
6. `WHERE` clauses with the `LIKE` operator (pattern matching), the wildcard characters are `_` (underscore) and `?`. Escaping of wildcard characters is supported as well.
7. Complex `WHERE` clauses with multiple conditions connected by `AND`, `OR` and `NOT`. Brackets must be used if AND and OR are mixed.

8. `GROUP BY` using the aggregation functions `SUM`, `MAX`, `MIN`, `COUNT(*)` and `AVG`
9. `HAVING`
10. `ORDER BY` with `ASC` and `DESC` direction modifiers.

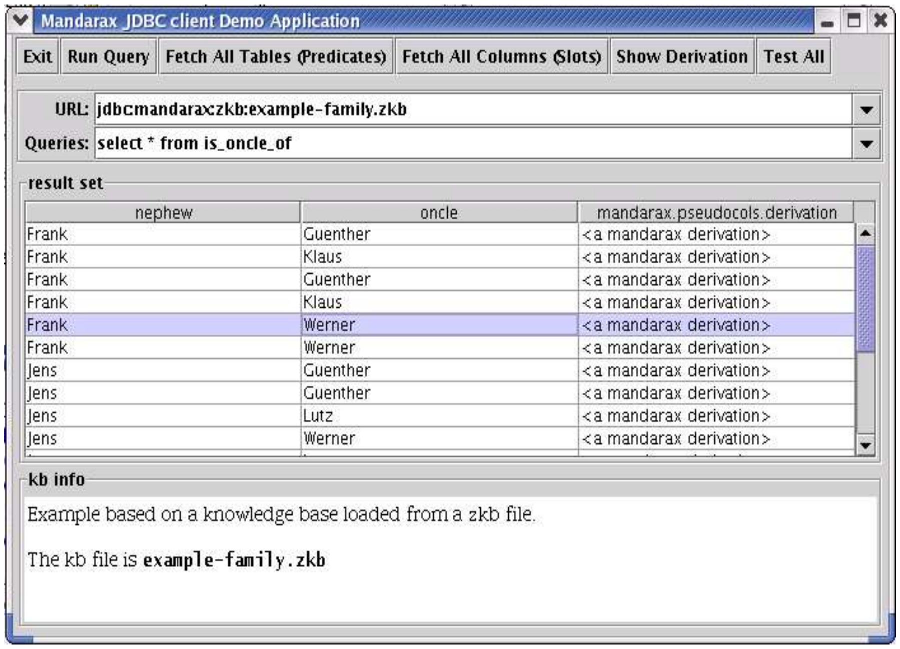
In particular, the following features are not (yet) supported:
1. Nested `SELECT` statements.
2. `IN` operator (use a `OR` group instead)
3. `BETWEEN` (use two simple comparison conditions instead)
4. Non aggregation functions such as `UPPER`, `LOWER` and string concatenation (`||`)
5. Pseudo columns such as `SYSDATE` and `USER`

Note that many database applications use `SELECT COUNT(*)` queries to estimate the time and the resources needed to answer a query. In case of mandarax, answering the `SELECT COUNT(*)` query is as expensive as answering the respective `SELECT` query itself and iterating over the entire result set!

The JDBC Example Application

The jdbc example package (`org.mandarax.examples.jdbc`) contains an executable class `MandaraxJdbcClientDemoApp`. This class is an ad hoc query tool that can be used in order to query mandarax knowledge bases using SQL. The tool supports the following features:

1. An editable URL field with a number of predefined URLs. Some of these URLs are based on (zkb, ruleml, xkb -) file. These files will be generated if necessary when the application is started.
2. Editable field for SQL queries, a list of predefined queries is available as well.
3. Database meta (tables and columns) data can be displayed as result sets.
4. The derivation pseudo column is supported. To display the derivation, select a row in the result set and press the `<Show Derivation>` button, or double-click on the row. Note that the derivation can be null (e.g., this is the case if the `SELECT` statement contains a `GROUP BY` clause).


Mandarax JDBC client Demo Application
Exit
Run Query
Fetch All Tables (Predicates)
Fetch All Columns (Slots)
Show Derivation
Test All
result set
kb info

URL: jdbc:mandarax:zkb:example-family.zkb

Queries: select * from is_oncle_of

nephew	oncle	mandarax.pseudocols.derivation
Frank	Guenther	<a mandarax derivation>
Frank	Klaus	<a mandarax derivation>
Frank	Guenther	<a mandarax derivation>
Frank	Klaus	<a mandarax derivation>
Frank	Werner	<a mandarax derivation>
Frank	Werner	<a mandarax derivation>
Jens	Guenther	<a mandarax derivation>
Jens	Guenther	<a mandarax derivation>
Jens	Lutz	<a mandarax derivation>
Jens	Werner	<a mandarax derivation>

Example based on a knowledge base loaded from a zkb file.

The kb file is **`example-family.zkb`**

Figure 3 The JDBC Client Demo Application

Using Mandarax JDBC with Generic Database Clients

The mandarax jdbc driver has been tested successfully with the following generic (JDBC based) database clients:

1. OpenOffice 1.1
2. SquirreL SQL client 1.1
3. DBVisualizer 3.3.1

Integrating mandarax is usually straight forward, it is important that all libraries used by mandarax are in the class path used by the respective client.

Miscellaneous

Logs

Mandarax supports comprehensive logging. The mandarax logging support consists of a hierarchy of log categories defined as constants in `org.mandarax.kernel.LogCategories`, several log levels (like "err or", "warn", "deb ug" and "info") and log appenders (like the console, text files, xml files, network resources or OS log targets). The mandarax log categories are:

```
1.  Log Categories the inference engine
    1.1. MANDARAX.IE
    1.2. MANDARAX.IE.LOOPCHECK
    1.3. MANDARAX.IE.RESULT
    1.4. MANDARAX.IE.STEP
    1.5. MANDARAX.IE.UNIFICATION
```

```
2.  Log Categories for the knowledge base
    2.1. MANDARAX.KB
    2.2. MANDARAX.KB.ADD
    2.3. MANDARAX.KB.REMOVE
    2.4. MANDARAX.KB.EVENT
    2.5. MANDARAX.KB.MOVE
3.  Log category for test cases
    3.1. MANDARAX.TESTS
4.  Log category for the SQL integration
    4.1. MANDARAX.SQL
5.  Log category for the logic factory
    5.1. MANDARAX.LF
6.  Log category for the XKB xml interface
    6.1. MANDARAX.XKB
7.  Log category for the JDBC driver
    7.1. MANDARAX.JDBC
```

The actual logging is delegated to a log service provider framework. These frameworks are very flexible, and can be configured by scripts, property files or xml files. In older mandarax versions (prior to 2.3.1) there was only one 'hard coded' log service provider, apache log4j. This has been redesigned using a design based on abstract loggers and logger factories. Factories for two log service providers, Apache Log4J and the JDK `java.util.logging` package, are part of the mandarax distribution. The default log service provider is selected according to the following rules:

1. Mandarax looks up the System property `org.mandarax.logger.` If there is such a system property and the property values is the name of a class implementing the LoggerFactory interface, an instance of this class will be used as default factory.
2. If `org.apache.log4j.Logger` is found in the class path, log4j will be used.
3. (else) if `java.util.logging.Logger` is found in the classpath (i.e., JDK version 1.4 or better is used), JDK logging will be used.
4. If none of the above rules can be applied, a 'dummy' log implementation that outputs all log entries on the console will be used.

Test Cases

The mandarax library contains numerous test cases organized in packages starting with “**`test`**”. These packages are not required in an application using mandarax! We use the well known JUnit framework for testing, each package contains one or many classes ending with “**`Tests`**”. These classes have a main method that invokes the (swing) junit test runner. The package `test.org.mandarax.testsupport` contains the class `TestAll`. This class is the “s uper test suite” that invokes the test runner for all mandarax tests. There are currently (Mandarax 3.0) **779** test cases in the test suite.

The test cases create numerous test files. These file are stored in the `tmptestdata` folder.

Modules

Mandarax modules consist of packages. There are 'depends on' relationships between modules. If no other modules depend on a certain module, the module packages can be safely removed from the mandarax distribution if a slim distribution is wanted and the respective functionality is not needed.

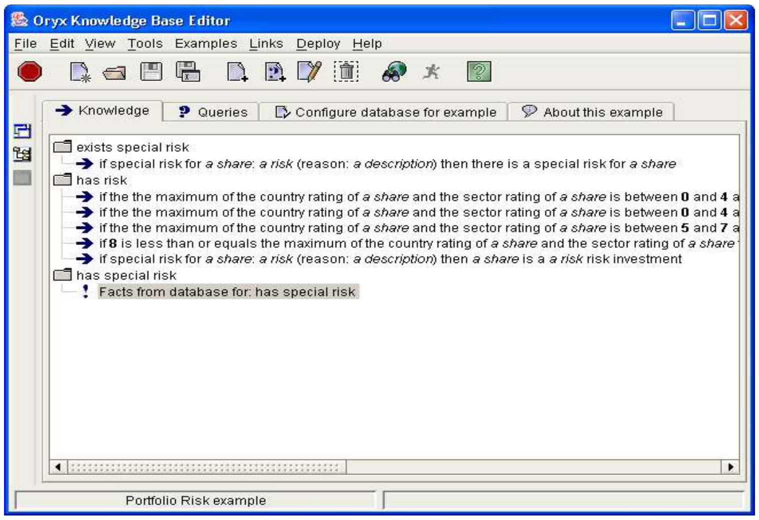
Module Name	*Packages*	*Description*	*Depends on*
kernel	kernel*, lib*, util*	Mandarax kernel	
reference	reference*	reference implementation	mandarax kernel
xkb	xkb*	XKB persistence/ serialization	mandarax kernel, sql
zkb	zkb*	ZKB persistence/ serialization	mandarax kernel, sql
jdbc	jdbc*	JDBC driver	mandarax kernel, zkb, xkb
sql	sql*	SQL predicates, clause sets and functions	mandarax kernel
tests	test	Test cases	mandarax kernel, sql, xkb, zkb
examples	examples*	Examples	tests

Extensions

Oryx

Oryx is a graphical front end for mandarax. It supports verbalization of knowledge using Swing and JSP based user interfaces. The design is modular and parts of oryx (for instance, editors for queries, single rules, etc) can be used separately in applications. The component models used are java beans / swing and jsp tag libraries, respectively. The oryx design is based on the flexible MVC/ MVC2 design pattern.

Conceptually, oryx adds the concept of a *repository*. A knowledge base is associated with a repository that contains “meta” information about predicates, functions, data sources, knowledge verbalization etc. Oryx contains a number of examples illustrating the integration of knowledge from SQL, JNDI and other sources.


Oryx Knowledge Base Editor
File Edit View Tools Examples Links Deploy Help
Knowledge
Queries
Configure database for example
About this example

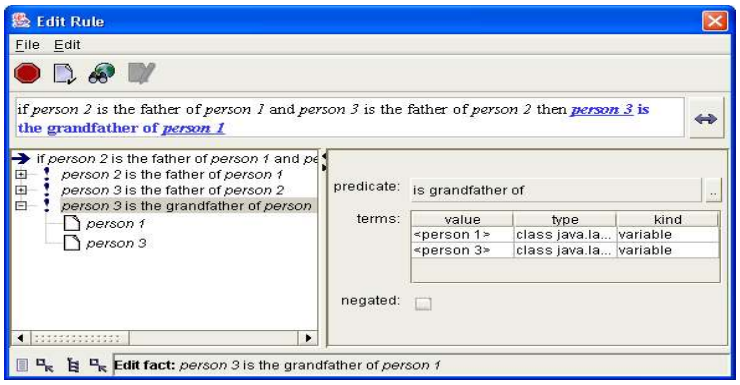
exists special risk
- if special risk for *a share*: *a risk* (reason: *a description*) then there is a special risk for *a share*

has risk
- if the the maximum of the country rating of *a share* and the sector rating of *a share* is between **0** and **4** a
- if the the maximum of the country rating of *a share* and the sector rating of *a share* is between **0** and **4** a
- if the the maximum of the country rating of *a share* and the sector rating of *a share* is between **5** and **7** a
- if **8** is less than or equals the maximum of the country rating of *a share* and the sector rating of *a share*
- if special risk for *a share*: *a risk* (reason: *a description*) then *a share* is a *a risk* risk investment

has special risk
- Facts from database for: has special risk

Portfolio Risk example

Figure 4 The Oryx Knowledge Base Editor



if *person 2* is the father of *person 1* and *person 3* is the father of *person 2* then ***person 3* is the grandfather of *person 1***

if *person 2* is the father of *person 1* and *pe*
- *person 2* is the father of *person 1*
- *person 3* is the father of *person 2*
- *person 3* is the grandfather of *person*
  - *person 1*
  - *person 3*

predicate: is grandfather of

terms:

value	type	kind
<person 1>	class java.la...	variable
<person 3>	class java.la...	variable

negated: ☐

Edit fact: *person 3* is the grandfather of *person 1*

Figure 5 The Oryx Rule Editor

Mandarax ECA

Mandarax ECA (**E**vent, **C**ondition, **A**ction) is an extension that can be used to program reactive agents. The system is event driven: events have registered event listeners (handlers), these listeners query the knowledge base for the next action that must be performed. Both the event and the action mechanism are designed for distributed systems. There is a Mandarax ECA portfolio agent example application that uses POP event sources and an SMTP actions. Java Messaging Service (JMS) would be a suitable infrastructure for real world applications based on Mandarax ECA. More details on Mandarax ECA can be found on the mandarax home page.

Appendix A – Required Libraries

name	version	URL	license	description
Apache log4j	1.1.3	`http://jakarta.apache.org/log4j`	Apache open source license	Log framework, see the class LogCategories for a list of log categories.
Apache commons collections	2.1	`http://jakarta.apache.org/commons`	Apache open source license	Data structures (collections and iterators)
jdom	1.0 beta 7	`www.jdom.org`	Apache-style open source license	Easy to use java xml interface, required for the XKB packages. Requires itself JASP 1.1
junit	3.7	`www.junit.org`	IBM public license	Test cases, not required for runtimes without thye test packages
javax.sql	JDK 1.4	`java.sun.com`	See JDK license	Not in (older) standard J2SE distributions but required by the XML package.

Table 5 3rd Party Libraries

Appendix B - Diagrams

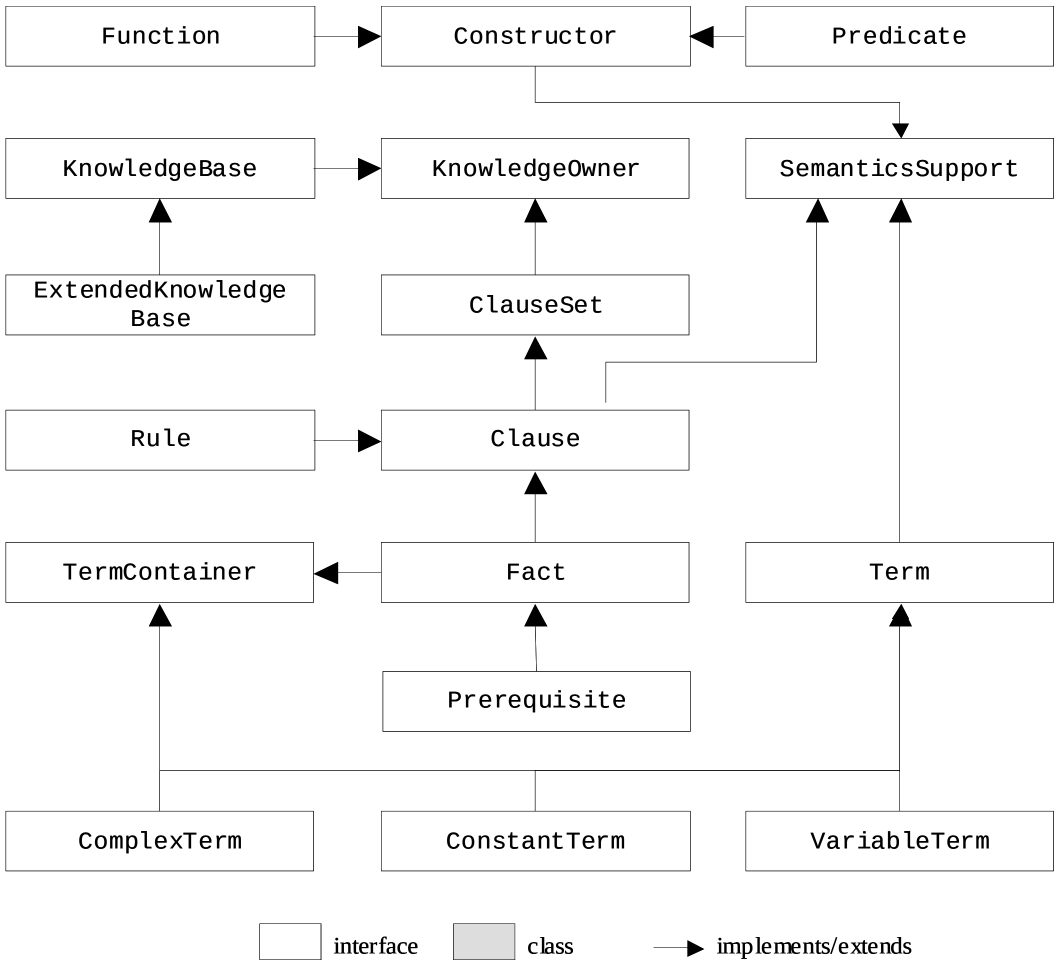


Figure 6: Core mandarax interfaces (kernel package) and their (extends) relationship

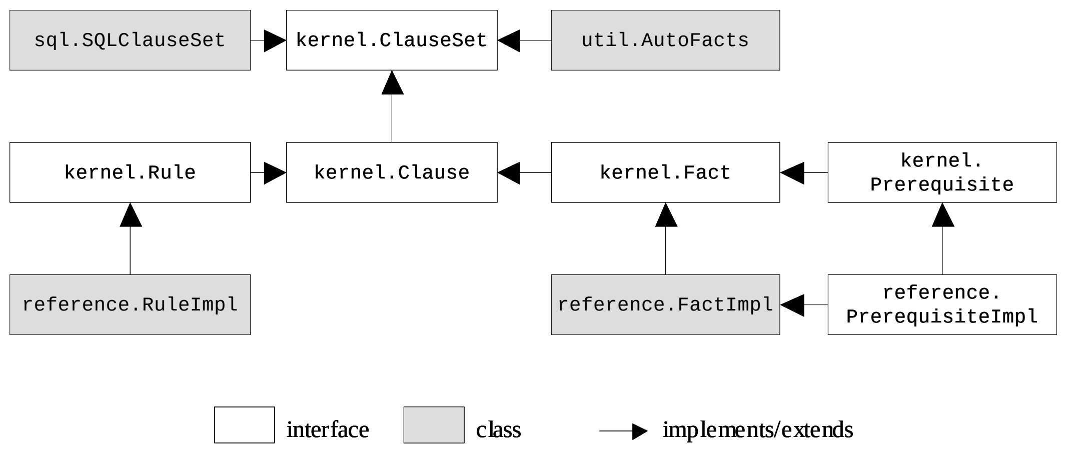


Figure 7: Mandarax Clauses and Clause Sets (package names without `org.mandarax` prefix)

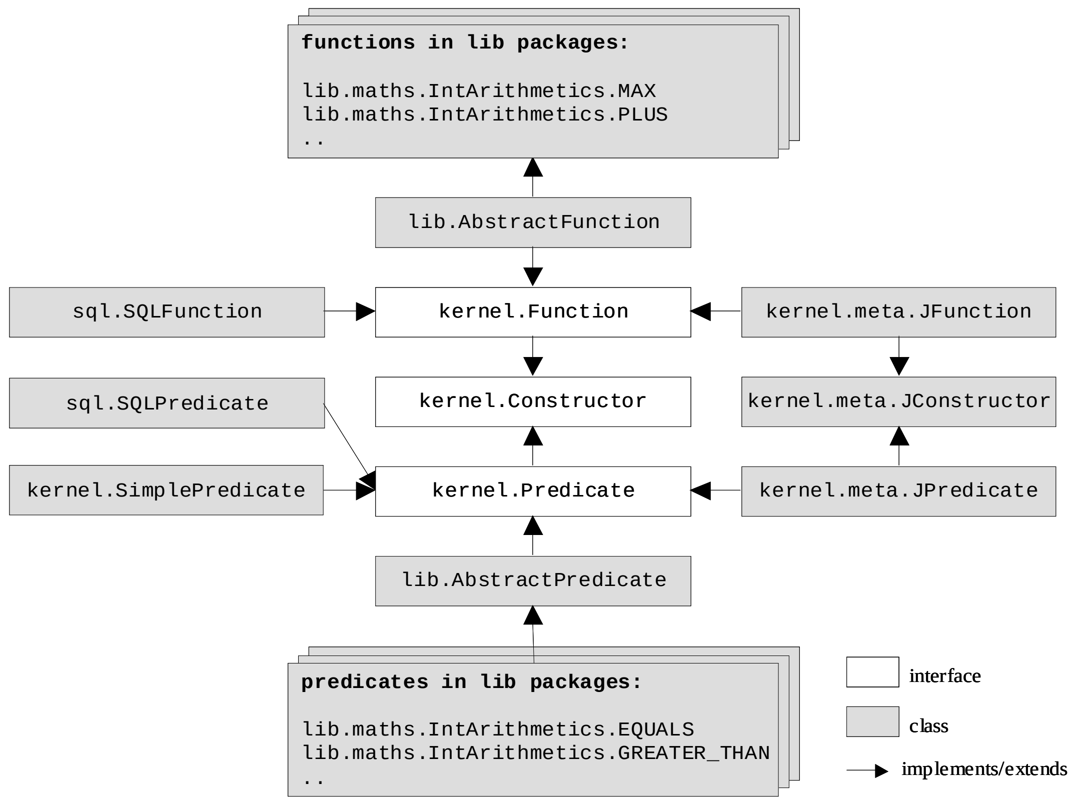


Figure 8: Mandarax Predicates and Functions (package names without `org.mandarax` prefix)

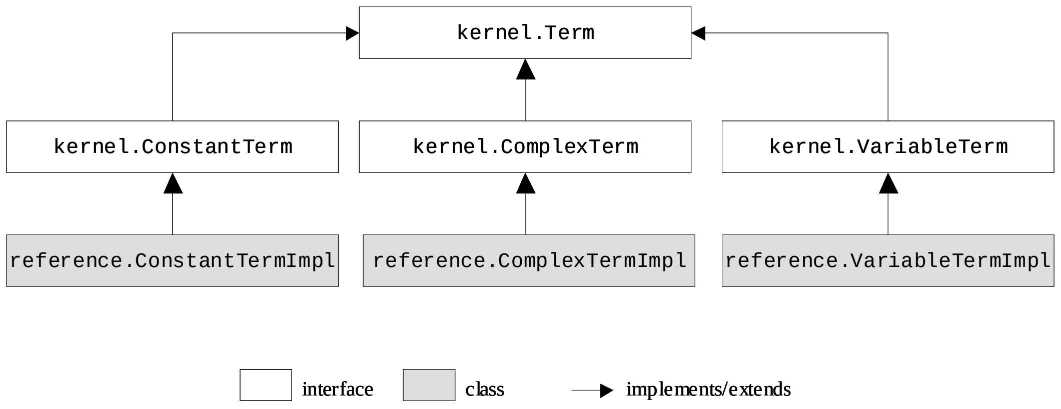


Figure 9: Mandarax Terms (package names without `org.mandarax` prefix)

Appendix C – Project History

Version	Date	Features
3.0	Nov 2003	1. New JDBC Driver incl test cases and a demo application 2. Result set filters with SQL like ORDER BY, GROUP BY and WHERE functionality 3. Minor bugfixes mainly in the ZKB and SQL packages 4. Support for names slots in predicates. 5. Knowledge bases expose the predicates used in knowledge within the knowledge base (new predicates() method).
2.2	12 Feb 2003	1. DynaBeanFunction 2. ZKB persistency module 3. New example using a relational database 4. Knowledge bases support comparators
2.1	23 Nov 2002	1. Cut 2. DefaultInferenceEngine 3. ResolutionInferenceEngine4 4. AdvancedKnowledgeBase redesigned
2.0	31 Oct 2002	1. Negation ("negation as failure) 2. ResolutionInferenceEngine3 3. Interface Prerequisite. 4. *Pluggable* semantic evaluation policy. 5. close() method in ClauseSetIterator , SQLClauseSet and SQLFunction 6. Some new methods in DerivationNode supporting a more detailed analysis of the derivation tree. 7. Clause sets and query support *additional properties*, i.e. simple key value associations. 8. A new XKB driver (2.0) supporting the respective new features (close connection flag, negation, properties).
1.9	18 Aug 2002	1. Bugfixes and testing related to serialization 2. Fully instantiated queries are now handled 3. A query result can also be complex term still containing variables 4. New interface ..kernel.Query and reference implementation ..reference.QueryImpl. 5. New rule ml driver supporting RuleML 0.8.1
1.8	18 Jun 2002	1. ResultSetFilter and CachedResultSet . 2. The old inference engine has been re-organized to share as much code as possible with the new implementation.
1.7	23 May 2002	1. SQLPredicate supports now perform. 2. New JDBC like query interface, including a (JDBC 2 like) ResultSet interface. 3. Implementation of this interface by the ResolutionInferenceEngine 4. Clause set method throw now a ClauseSetException. 5. The query methods in InferenceEngine throw now an InferenceException. 6. Inference engines can decide how to handle situation when looping over a clause set leads to an exception.

1.6	3 Mar 2002	1. Migrated to log4j 1.1.3 2. Updated to JUnit 3.7 3. Uses Ant. 4. New package org.mandarax.xkb.framework with a modular framework for XML interfaces, XKB driver supporting almost all mandarax features is part of this package. 5. Functions based on SQL queries. 6. Systematic support for semantic evaluation in reference inference engine. 7. New interface org.mandarax.kernel.SemanticsSupport.
1.5	27 Nov 2002	New package org.mandarax.sql
1.4	13 Aug 2002	1. Integration of xkb package (XML interface). 2. New lib packages integrating standard functionality for arithmetic, strings and dates.
1.3	4 Mar 2002	1. Plugin architecture + reference implementation of a loop checking algorithm. 2. Support for rules with prerequisites connected by *OR* . 3. Causes keep now a reference to their clause sets. 4. Support for new (org.apache) log4j package names.
1.2	3 Dec 2000	1. New package org.mandarax.math. 2. Test cases support the current version of the junit test framework with junit.* package names (instead of test.* package names). 3. New knowledge base implementation org.mandarax.reference.AdvancedKnowledgeBase. 4. New package org.mandarax.examples.crm containing a comprehensive example how to calculate discounts using a knowledge base and plain data. to start the applet. 5. The package org.mandarax.demo has been renamed to org.mandarax.examples.family. 6. New class org.mandarax.util.AutoFacts. 7. New utility class org.mandarax.util.LogicFactorySupport. 8. New utility class org.mandarax.util.ProofAnalyzer.
1.1	14 Jun 2000	1. Fact, Rule, ComplexTerm, VariableTerm and ConstantTerm are now interfaces, implementations of these interfaces are provided in the org.mandarax.reference package, instances are now created using a factory. 2. Log support using the library log4j 3. Clause sets fire events to notify listeners about changes 4. Knowledge bases fire events to notify listeners about changes 5. New interface ExtendedKnowledgeBase. 6. Reference inference engine plugin for unification algorithms, by default Robinson's algorithm is used 7. Reference inference engine plugin for selection policies, two policies (right most and left most) are provided 8. Extended support to use the java object model (new methods resolve() in Fact and Term and perform() in Function and Predicate 9. Many classes are now serializable in order to support technologies like persistency using serialization, RMI and Enterprise Beans 10. Performance improvements

1.0	1 Jan 2000	Project Launch.

Alphabetical Index

Bibliography

[DKSW 03] J. Dietrich , A. Kozlenkov , M. Schroeder , G. Wagner: Rule-based agents for the semantic web. Electronic Commerce Research and Applications 2 (2003) 323–338.

[Grand 98] M. Grand: Patterns in Java. Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML. Wiley 1998.

[Wag 98] G. Wagner: Foundations of Knowledge Systems: With Applications to Databases and Agents. Kluwer Academic Publishers 1998.