The ROSETTA C++ Library: Overview of Files and Classes*

Aleksander Øhrn†

Contents

*This document is under perpetual development. Last updated March 22, 2000.

†Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway. E-mail: aleks@idi.ntnu.no.

List of Figures

List of Tables

1 Introduction

This document describes the ROSETTA C++ library [15], a collection of C++ classes and functions developed as part of my doctoral dissertation [12]. In addition to providing a useful collection of algorithms and data structures specific to discernibility-based data mining, the library also contains a lot of support routines, infrastructure and classes relevant for machine learning in general. The ROSETTA C++ library is not tied up to any particular hardware platform, operating system or compiler.

Miscellaneous tips and take-home points are throughout this document indented and marked with a special symbol. An example, which incidentally is also a real tip, is the following:

TIP All files in the ROSETTA C++ library are best viewed with tab size 2. This preserves the intended alignment and indentation in the source code, and makes the code a lot easier to read. Using a syntax-highlighting editor is also strongly recommended.

After you have read this document, browsed the source code and otherwise acquainted yourself with the ROSETTA C++ library, these are the steps needed to get things up and running:

1. Write a *main* routine that specifies what you want your program to do. This is further described in Section 2.2.

2. Create a *make* file that specifies compiler settings, which files are to be compiled as part of your project, and dependencies between these. This is further described in Section 3.

The list of files covered by this document is not exhaustive. Rather, this document only intends to give an overview of the overall library structure and the contents of some of the most central files. The source code is fairly well interspersed with comments, and the reader is encouraged to browse the source code for further documentation.

TIP Generally, there is one C++ class per file, with the name of the file being the same as the name of the class set in lowercase. For example, the file `foobar.*` would contain source code for the class *FooBar*.

This document does not provide detailed documentation of the inner workings of all the algorithms in the ROSETTA system. For this, see [12,13] and references therein, as well as the source code itself.

In Section 2, an overview of the directories and files of the ROSETTA C++ library is given. A graphical depiction of the library's directory structure can be obtained from this document's table of contents. Some comments on how to configure your compiler are given in Section 3, while miscellaneous items not discussed elsewhere are featured in Section 4. Appendix A lists input/output type signatures for algorithmic classes. C++ class hierarchies are presented in Appendix B, graphically depicting the inheritance relationships between classes.

2 Directories and Files

2.1 ⟨`stdafx.*`⟩

This file is basically an include file for standard system include files, or project specific include files that are used frequently, but are changed infrequently. You will rarely, if ever, change this file.

Under Microsoft Visual C++, precompiled headers are typically used through this file.[1]

2.2 ⟨`main.*`⟩

This file you have to write yourself. The *main* routine is your "driver" program, specifying what your program should do and in what capacity you use the ROSETTA C++ library.

For details and examples of *main* routines, see [15].

2.3 ⟨`kernel`⟩

Contains the computational kernel of ROSETTA.

2.3.1 ⟨`kernel/basic`⟩

Contains elementary structures, e.g., smart pointers and classes for reference counting, basic structures such as strings, bit sets, vectors and maps, macro definitions, management of identifiers, etc.

2.3.1.1 ⟨`kernel/basic/array.*`⟩ Vector template that supports dynamic resizing. Currently, this template can only be used for primitive types, or types that provide both an empty constructor (i.e., are "default constructible") and an assignment operator (i.e., are "assignable").

TIP This class is not in use by default, but it can be used as a drop-in replacement for STL vectors, e.g., if we want to introduce memory pooling techniques to reduce memory fragmentation. This is controlled through ⟨`kernel/basic/vector.*`⟩ and macros in ⟨`kernel/basic/macros.*`⟩.

2.3.1.2 ⟨`kernel/basic/bits.*`⟩ Implements a vector of bits. Provides a compact representation of the set $\{0, \ldots, n-1\}$, useful if n is not too high and the average set cardinality is not too low. (Otherwise, consider using an integer vector instead.) Offers efficient methods for set union, intersection, difference, subset testing, etc.

2.3.1.3 ⟨`kernel/basic/handle.*`⟩ Provides a "smart pointer" to objects derived from the *Referent* class. The advantage of using handles instead of standard C++ pointers is that the memory management of large objects is significantly simplified and the safety of memory operations is increased.

TIP If your compiler supports member templates in a robust fashion, see Section 3 for a description of the *_MEMTMPL* flag.

[1]Under Microsoft Visual C++ 6.0, the use of precompiled headers is controlled from the *C/C++* tab in the dialog box that appears if you select the *Project/Settings...* menu entry. Select the *Precompiled Headers* category.

2.3.1.4 ⟨`kernel/basic/identifier.*`⟩ Provides run-time type identification through the methods *IsA* and *GetId*. *IsA* defines a transitive relation on types, while *GetId* returns the most specific type. Data on known types are administered by the *IdHolder* class.

TIP Classes derived from *Identifier* should use the *DECLAREIDMETHODS* and *IMPLEMENTIDMETHODS* macros to declare and implement the *IsA* and *GetId* methods.

2.3.1.5 ⟨`kernel/basic/idholder.*`⟩ Static class dealing with the administration of type information used by the *Identifier* class, e.g., class names and class descriptions.

TIP See ⟨`kernel/basic/ids.*`⟩ for how new types are registered, using the *DECLAREID* and *IMPLEMENTID* macros.

2.3.1.6 ⟨`kernel/basic/ids.*`⟩ This is where all the type information for classes derived from *Identifier* is declared. The type information is stored by the *IdHolder* class.

TIP It is the information entered here that defines menu texts in the ROSETTA GUI, as well as the names of algorithms that are recognized in ROSETTA command scripts.

2.3.1.7 ⟨`kernel/basic/interval.*`⟩ Represents an interval over the set of reals. The interval may be open or closed, and finite or infinite. Offers convenient methods for, e.g., checking whether a value is inside an interval.

2.3.1.8 ⟨`kernel/basic/map.*`⟩ Functions as a portability layer. Currently includes the associative map template from the STL library, wrapped in the *Map* macro from ⟨`kernel/basic/macros.*`⟩.

2.3.1.9 ⟨`kernel/basic/macros.*`⟩ This is where system-wide macros are defined. Contains macros for declaring and implementing identifiers and run-time type methods, installing prototype objects, defining containers, casting, etc.

2.3.1.10 ⟨`kernel/basic/memory.*`⟩ If you want to overload global memory allocation/deallocation operators, this is the place to do it.

2.3.1.11 ⟨`kernel/basic/message.*`⟩ The *Message* class relays general messages to the user, as well as warning and error messages. Proper indentation of messages is automatically taken care of via a technique for counting the number of currently instantiated *Message* objects. This class also offers a cancellation feature for the "topmost" *Message* object, so that lengthy computations can be aborted by the user. Employs the static *MessageHelper* class as a helper class.

2.3.1.12 ⟨`kernel/basic/persistent.*`⟩ Base class for persistent objects, i.e., objects that offer *Load* and *Save* methods.

2.3.1.13 ⟨`kernel/basic/referent.*`⟩ The top level *Referent* class works in close cooperation with the *Handle* template. Together, they automate reference counting and object deletion. The *Referent* class overloads the *new* and *delete* operators.

2.3.1.14 ⟨`kernel/basic/string.*`⟩

String, a general-purpose string class, is used throughout the library. Offers a set of convenient methods for parsing and concatenating text, converting between text and numbers, etc.

The *String* class is implemented using reference counting, meaning that multiple strings may share the same physical representation. This makes string assignments very cheap.

2.3.1.15 ⟨`kernel/basic/types.*`⟩

Some fundamental types are defined here, e.g., *Id* and possibly also *bool* in case of older compilers.

2.3.1.16 ⟨`kernel/basic/undefined.*`⟩

The static *Undefined* class defines a scope around "magic" constants that indicate undefined values.

2.3.1.17 ⟨`kernel/basic/vector.*`⟩

Functions as a portability layer. Currently includes the vector template from the STL library, wrapped in the *Vector* macro from ⟨`kernel/basic/macros.*`⟩.

2.3.2 ⟨`kernel/system`⟩

Contains wrappers for standard C system files. Functions as a portability layer.

2.3.2.1 ⟨`kernel/system/sys`⟩

Contains wrappers for certain standard C system files. Functions as a portability layer.

2.3.2.2 ⟨`kernel/system/stl`⟩

Contains the STL[2] implementation in use.[3] Implementations of basic containers such as vectors, maps and sets, etc., and some algorithms that operate on these via iterators.

TIP STL containers should never be included directly from this directory. Rather, portability layers like, e.g., ⟨`kernel/basic/vector.*`⟩ should be included instead.

2.3.3 ⟨`kernel/structures`⟩

Contains code for higher-level structures such as decision tables, collections of reducts and rules, discernibility matrices and functions, ROC curves, etc.

2.3.3.1 ⟨`kernel/structures/annotatedstructure.*`⟩

Structural objects that have an *Annotation* object associated with them inherit from *AnnotatedStructure*. Typically, only objects that a user will interact with directly in a user interface need to be annotated.

2.3.3.2 ⟨`kernel/structures/annotation.*`⟩

Holds annotation data for annotated structures. An annotation holds the object's name, physical location, a history log and a general comment field.

2.3.3.3 ⟨`kernel/structures/approximation.*`⟩

Represents a rough set approximation of a set of objects. Offers methods for accessing the lower and upper approximations, the boundary and outside regions, etc.

[2]Standard template library.

[3]Currently STLport [16], an adaptation of Silicon Graphics' STL where great care has been taken to make the templates as portable and robust as possible against compiler bugs and platform quirks.

2.3.3.4 ⟨`kernel/structures/attribute.*`⟩ Base class for *Attribute* objects, used by the *Dictionary* class. Functions as a map between coded integer values and actual textual or numerical values.

2.3.3.5 ⟨`kernel/structures/batchclassification.*`⟩ Summarizes the result of having classified a set of objects in a decision table. Holds a *ConfusionMatrix* object and possibly ROC data, too. Output by algorithms in the *BatchClassifier* family.

2.3.3.6 ⟨`kernel/structures/binaryoutcomecurve.*`⟩ Base class for curves generated from binary outcome classification data.

2.3.3.7 ⟨`kernel/structures/booleanfunction.*`⟩ Base class for Boolean POS or SOP functions. Offers methods for simplification of such functions, i.e., removal of duplicates and absorption.

Internally, the *BooleanFunction* class is implemented as a vector of *Bits* pointers, with one *Bits* "component" per sum or product. Each component" can have a weight associated with it. Such weights may reflect, e.g., a measure of importance or simply an occurrence count. Since *Bits* pointers are used, a "component' may be shared between a *BooleanFunction* object and, e.g., a *DiscernibilityMatrix* object. Eliminating duplication of data also speeds up certain operations.

2.3.3.8 ⟨`kernel/structures/booleanposfunction.*`⟩ Specialization of *BooleanFunction* for POS functions. Each function "component" is thus interpreted as a Boolean sum.

2.3.3.9 ⟨`kernel/structures/booleansopfunction.*`⟩ Specialization of *BooleanFunction* for SOP functions. Each function "component" is thus interpreted as a Boolean product.

2.3.3.10 ⟨`kernel/structures/calibrationcurve.*`⟩ Represents a calibration curve, constructed from a number of pairs generated from a binary outcome classification procedure. Also offers methods for computing, e.g., the Brier score and its covariance decomposition [1].

2.3.3.11 ⟨`kernel/structures/classification.*`⟩ Summarizes the result of classifying a single object in a decision table. A classification result can be seen as a list of possible decision values along with some extra information, e.g., a measure of evidence or certainty associated with each suggested decision value. Output by algorithms in the *Classifier* family.

2.3.3.12 ⟨`kernel/structures/confusionmatrix.*`⟩ Represents a square matrix with integer entries. Used to summarize the results of a batch classification procedure.

2.3.3.13 ⟨`kernel/structures/decisiontable.*`⟩ The *DecisionTable* class represents a flat data table. Some of the rows or columns can be "masked", i.e., made invisible to *DecisionTable* clients. Columns can be marked as having condition or decision status.

Internally, all table entries are integers. The conversion between these coded values and the actual values as perceived by the user is taken care of by an associated *Dictionary* object.

TIP For specialized applications with very large tables where all table entries are either 0 or 1, *DecisionTable* could be subclassed to use a vector of *Bits* as its internal representation. For large tables, this could save significant amounts of memory.[4]

[4]On most current platforms, an *int* occupies four bytes. Representing a row with a *Bits* object instead of a vector of integers would thus result in a memory savings factor of 32.

2.3.3.14 ⟨`kernel/structures/decisiontables.*`⟩ Represents a collection of *DecisionTable* objects. Rarely in use.

2.3.3.15 ⟨`kernel/structures/dictionary.*`⟩ The *Dictionary* class converts between real-world values and the coded integer values used as the internal representation. Works in cooperation with *DecisionTable* objects. A *Dictionary* object is composed of a set of *Attribute* objects.

2.3.3.16 ⟨`kernel/structures/discernibilityfunction.*`⟩ Represents a Boolean POS function constructed from a decision table.

2.3.3.17 ⟨`kernel/structures/discernibilitymatrix.*`⟩ Represents a discernibility matrix, where each entry is a pointer to a *Bits* object. Empty entries are represented by *NULL* pointers.

A client may perceive this as a full matrix with dimensionality equal to the cardinality of the universe of objects. Internally, however, less than half the matrix is stored. We only need one representative per equivalence class, and we additionally exploit that the matrix is symmetric and has an empty diagonal. The mapping between the full "logical" view of the matrix and the reduced internal representation is done in $O(1)$ time.

2.3.3.18 ⟨`kernel/structures/equivalenceclass.*`⟩ Represents a single equivalence class, viewed as a set of object indices into some ancestral *DecisionTable* object. Additionally offers some query methods.

2.3.3.19 ⟨`kernel/structures/equivalenceclasses.*`⟩ Represents a collection of *EquivalenceClass* objects. Employed by, e.g., the *Approximation* and *Partitioner* classes.

2.3.3.20 ⟨`kernel/structures/floatattribute.*`⟩ Specialization of *Attribute* for attributes where the values are floating point numbers. Maps between integers and floats by multiplying by a factor of $10^{\pm n}$, where n is a scaling exponent.

2.3.3.21 ⟨`kernel/structures/generalizeddecision.*`⟩ Represents a generalized decision value, i.e., a collection of decision values. Their corresponding cardinalities are also stored, thus enabling computation of probabilities.

2.3.3.22 ⟨`kernel/structures/graph.*`⟩ Represents a directed graph with vertices taking labels from a finite domain of integers, not necessarily $\{0, \ldots, n-1\}$. Implemented by an adjacency matrix, best for small or dense graphs. Vertices and edges are typically specified in a file, and a small language for this purpose is supported. Various graph algorithms such as Dijkstra's, Floyd's and Warshall's algorithms are available.

2.3.3.23 ⟨`kernel/structures/history.*`⟩ Represents a history log. Used by the *Annotation* class.

2.3.3.24 ⟨`kernel/structures/historyentry.*`⟩ Represents an entry in a history log. Records who did what when. Used by the *History* class.

2.3.3.25 ⟨`kernel/structures/indiscernibilitygraph.*`⟩ Specialization of *Graph* for indiscernibility graphs. Offers methods to create such graphs from *Decisiontable* or *DiscernibilityMatrix* objects. Provides support for IDGs[5] via the *Discerner* class.

[5]Indiscernibility definition graphs. See [12] for details.

2.3.3.26 ⟨`kernel/structures/informationvector.*`⟩ Represents a vector of attribute/value pairs for an object in a decision table.

2.3.3.27 ⟨`kernel/structures/integerattribute.*`⟩ Specialization of *Attribute* for attributes where the values are integers. Performs an identity mapping.

2.3.3.28 ⟨`kernel/structures/ksdecisiontable.*`⟩ A simple specialization of *DecisionTable* that can be used if the system is compiled without the RSES library. Functions as a drop-in replacement for the corresponding RSES wrapper.

2.3.3.29 ⟨`kernel/structures/ksinformationvector.*`⟩ A simple specialization of *InformationVector* that can be used if the system is compiled without the RSES library. Functions as a drop-in replacement for the corresponding RSES wrapper.

2.3.3.30 ⟨`kernel/structures/ksreduct.*`⟩ A simple specialization of *Reduct* that can be used if the system is compiled without the RSES library. Functions as a drop-in replacement for the corresponding RSES wrapper.

2.3.3.31 ⟨`kernel/structures/ksrule.*`⟩ A simple specialization of *Rule* that can be used if the system is compiled without the RSES library. Functions as a drop-in replacement for the corresponding RSES wrapper.

2.3.3.32 ⟨`kernel/structures/parentstructure.*`⟩ Represents a structural object that can have other structural objects derived from it by means of some algorithmic object. The *ParentStructure* class provides a list of pointers to child *Structure* objects.

Note the difference between this class and the *Structures* class. This class functions as a base class for classes that have children pointers. The *Structures* class functions as a base class for classes that are conceived as sets of structures.

2.3.3.33 ⟨`kernel/structures/project.*`⟩ Represents a top level project, consisting of a tree of *Structure* and *Algorithm* objects. Offers methods to extract all objects of certain types from the tree.

2.3.3.34 ⟨`kernel/structures/projectmanager.*`⟩ Static class representing the pool of all *Project* objects that currently exist. The sole reason for having this class is so that the *FindParent* method at the *Structure* level works.[6] This enables us to track back to ancestral structures.

TIP The static *ProjectManager* class could perhaps be made transparent or "invisible' to the library user by inserting proper calls in the constructors and destructor of the *Project* class. Currently, however, this has to be done manually.

2.3.3.35 ⟨`kernel/structures/reduct.*`⟩ Base class for objects representing a reduct. A *Reduct* object can be viewed as a collection of attribute indices into a *DecisionTable* object.

2.3.3.36 ⟨`kernel/structures/reducts.*`⟩ Specialization of *Structures* for a collection of *Reduct* objects.

[6]Remember that the *Handle* mechanism only works as intended if there are no referential cycles. Since the *ParentStructure* class has pointers to its children, having a pointer back to its parent would introduce such cycles. The static *ProjectManager* class enables us to implement *FindParent* through a massive forward search instead. Unless the projects are very large, this works reasonably well.

2.3.3.37 ⟨`kernel/structures/roccurve.*`⟩ Represents an ROC curve, constructed from a number of pairs generated from a binary outcome classification procedure. The area under the curve can either be computed using trapezoidal integration or via the c-index.

2.3.3.38 ⟨`kernel/structures/rule.*`⟩ Base class for rules. The antecedent of a rule is a conjunction of descriptors, while the rule's consequent is a disjunction (or more precisely, a set known as the generalized decision value). If the consequent is missing then the rule is not strictly a rule, but is then often denoted a "pattern" or "template".

TIP The class design is currently not as clean as it should be, mainly due to historical reasons and to make integration with legacy code simpler. See ⟨`kernel/structures/rule.*`⟩ for details.

The attribute indices and values in a descriptor use the same encoding as the *DecisionTable* object from which the *Rule* object stems. For proper formatting, therefore, access to the *DecisionTable* object (and its associated *Dictionary* object) is required.

2.3.3.39 ⟨`kernel/structures/rulebasedclassification.*`⟩ Specialization of *Classification* for results obtained from applying algorithms in the *RuleBasedClassifier* family.

2.3.3.40 ⟨`kernel/structures/rules.*`⟩ Specialization of *Structures* for a collection of *Rule* objects.

2.3.3.41 ⟨`kernel/structures/stringattribute.*`⟩ Specialization of *Attribute* for attributes where the values are general strings. Translates between integers and strings by looking up stuff in associative maps.

2.3.3.42 ⟨`kernel/structures/structure.*`⟩ Base class for structural objects, i.e., objects that represent some kind of data structure. For reasons of interface uniformity, the *Structure* class offers a wide variety of virtual methods for, e.g., child and member structure management. (The actual implementation of these methods are found in subclasses of *Structure*. See Appendix B for class diagrams.) The *Structure* class also offers parent navigation through the family of *FindParent* methods.[7]

2.3.3.43 ⟨`kernel/structures/structures.*`⟩ Represents a set of *Structure* objects. This class is usually subclassed to cater for more specialized collections of structural objects.

2.3.4 ⟨`kernel/algorithms`⟩

Contains code for higher-level algorithms such as algorithms for discretization, computation of reducts, filtering of reducts and rules, voting, import/export routines, etc.

2.3.4.1 ⟨`kernel/algorithms/algorithm.*`⟩ Base class for algorithmic objects, i.e., objects designed to alter, create or process data structures. An *Algorithm* object takes a *Structure* object as input to its *Apply* method, and returns a *Structure* object if the application went well. The output structure may be identical to the input structure (modified or not), or a new structure of some type.

A general way of passing simple algorithm parameters is provided through the *SetParameter* method. The *IsApplicable* method identifies the supported input type.[8]

[7] See the description of *ProjectManager* for details.
[8] See Appendix A for details on this.

2.3.4.2 ⟨`kernel/algorithms/approximator.*`⟩ An *Approximator* algorithm takes as input a *DecisionTable* object, computes a rough set approximation, and returns an *Approximation* object.

2.3.4.3 ⟨`kernel/algorithms/batchclassifier.*`⟩ A *BatchClassifier* algorithm takes as input a *DecisionTable* object, attempts to classify all objects in the table using an algorithm in the *Classifier* family, and returns a *BatchClassification* object. ROC curves, calibration curves and log files can also be generated.

2.3.4.4 ⟨`kernel/algorithms/binarysplitter.*`⟩ Vertically splits a *DecisionTable* object in two parts, where the objects are randomly sampled. The relative sizes of the two parts can be specified. The two resulting *DecisionTable* objects are either attached to the input *DecisionTable* object as children, or returned in a compound *DecisionTables* objects.

2.3.4.5 ⟨`kernel/algorithms/brorthogonalscaler.*`⟩ A straightforward but not terribly efficient implementation of the supervised and multivariate discretization algorithm of Nguyen and Skowron [11]. Constructs a Boolean function[9] that expresses the discernibility present in the input *DecisionTable* object, and computes a prime implicant of this function. Employs the *NaiveScaler* and *JohnsonReducer* algorithms as aids.

2.3.4.6 ⟨`kernel/algorithms/classifier.*`⟩ Base class for algorithms that take as input an *InformationVector* object and return a *Classification* object, constructed using some classification scheme.

2.3.4.7 ⟨`kernel/algorithms/combinatorialcompleter.*`⟩ Specialization of *Completer*, where the input *DecisonTable* is completed through combinatorial expansion. Each row with missing values is expanded into two or more rows, so that all combinations of possible values for the empty entries are covered.

2.3.4.8 ⟨`kernel/algorithms/completer.*`⟩ Base class for table completion algorithms, i.e., algorithms that somehow produce a *DecisionTable* object with no missing values when fed with a *DecisionTable* object with missing values.

TIP Currently, completion algorithms do not save details of the completion process to file. They should, though, if relevant, since we could then later apply the same substitution values to another table.

2.3.4.9 ⟨`kernel/algorithms/conditionedcombinatorialcompleter.*`⟩ Specialization of *CombinatorialCompleter*, where the set of possible values for an object is conditioned to the decision class to which the object belongs.

2.3.4.10 ⟨`kernel/algorithms/conditionedcompleter.*`⟩ Base class for *Completer* algorithms that condition the completion values on the decision classes. Splits the input table into one table per decision class, does an unconditional completion on each of these, and combines them back into a single output table.

2.3.4.11 ⟨`kernel/algorithms/conditionedmeancompleter.*`⟩ Specialization of *MeanCompleter*, where the computed mean/mode values for an object are conditioned to the decision class to which the object belongs.

2.3.4.12 ⟨`kernel/algorithms/costinformation.*`⟩ Helper class that keeps information about attribute costs. Cost information is loaded from file. Support for shared costs is not yet implemented.

[9]This is the bottleneck in the current implementation.

2.3.4.13 ⟨`kernel/algorithms/cppruleexporter.*`⟩ Exports a *Rules* object to C++ code that realizes a classifier. Conflict resolution among firing *Rule* objects is done via standard voting.

2.3.4.14 ⟨`kernel/algorithms/cvserialexecutor.*`⟩ Specialization of *SerialExecutorLoop* where the sampling scheme is overloaded to implement *n*-fold cross-validation.

2.3.4.15 ⟨`kernel/algorithms/decisiontableexporter.*`⟩ Base class for algorithms that export some aspect of a *DecisionTable* object to some file format.

2.3.4.16 ⟨`kernel/algorithms/decisiontableimporter.*`⟩ Base class for algorithms that import *DecisionTable* objects from some alien format. The input *DecisionTable* is returned, filled with the contents of the file.

2.3.4.17 ⟨`kernel/algorithms/dictionaryexporter.*`⟩ Exports a *Dictionary* object to a format that can be edited and subsequently read back into the system using the *DictionaryImporter* algorithm.

2.3.4.18 ⟨`kernel/algorithms/dictionaryimporter.*`⟩ Complements the *DictionaryExporter* algorithm, described above.

2.3.4.19 ⟨`kernel/algorithms/discernibilityexporter.*`⟩ Base class for algorithms that export some discernibility aspect of *DecisionTable* objects. Provides support for IDGs. Optionally, a masked attribute can be used to name the exported data.

2.3.4.20 ⟨`kernel/algorithms/discernibilityfunctionexporter.*`⟩ Specialization of *DiscernibilityExporter*. Computes and exports one or more discernibility functions, simplified or not.

2.3.4.21 ⟨`kernel/algorithms/entropyscaler.*`⟩ Implements the discretization algorithm of Dougherty et al. [4]. Recursive algorithm based on entropy considerations.

2.3.4.22 ⟨`kernel/algorithms/equalfrequencyscaler.*`⟩ Implements equal frequency discretization, a simple unsupervised and univariate discretization algorithm. Fixing a number n and examining the histogram of each attribute, $n - 1$ cuts are determined so that approximately the same number of objects fall into each of the n bins.

2.3.4.23 ⟨`kernel/algorithms/executor.*`⟩ Base class for algorithms that execute command scripts.

2.3.4.24 ⟨`kernel/algorithms/exporter.*`⟩ Base class for algorithms that export some aspect of structural objects in some format. Most, if not all, algorithms in the *Exporter* family return the input structure unmodified. That is, they are simple pass-through operations where the exporting is a side-effect.

2.3.4.25 ⟨`kernel/algorithms/filter.*`⟩ Base class for algorithms that remove individual members from collections on the basis of some filtering criterion.

2.3.4.26 ⟨`kernel/algorithms/holte1rreducer.*`⟩ Returns all singleton attribute sets, inspired by the paper of Holte [8]. 1R rules are attached to the returned *Reducts* object as a child *Rules* object.

2.3.4.27 ⟨`kernel/algorithms/htmlreporter.*`⟩ Specialization of *Reporter* supporting HTML format. The exported report contains hyperlinks.

2.3.4.28 ⟨`kernel/algorithms/importer.*`⟩ Base class for algorithms that import structural objects from some medium. Typically, the input *Structure* is emptied, filled and returned.

2.3.4.29 ⟨`kernel/algorithms/indiscernibilitygraphexporter.*`⟩ Specialization of *DiscernibilityExporter*. Exports a graph that can be used to visualize the indiscernibility relation. The graph is exported in a format recognized by the GraphViz suite of layout tools [5].

2.3.4.30 ⟨`kernel/algorithms/johnsonreducer.*`⟩ Specialization of *Reducer*, implementing a variation of the set covering heuristic of Johnson [9]. The greedy algorithm is reasonably fast, but computes only a single prime implicant of a discernibility function. The algorithm has a bias towards finding a solution of minimal length. Approximate solutions are supported.

2.3.4.31 ⟨`kernel/algorithms/keyword.*`⟩ The static *Keyword* class defines a scope around "magic" strings that are used in the *Algorithm* methods *SetParameter* and *GetParameters*.

2.3.4.32 ⟨`kernel/algorithms/kidnapper.*`⟩ Specialization of *ScriptAlgorithm*. Returns child number i attached to the input structure.

2.3.4.33 ⟨`kernel/algorithms/ksrulegenerator.*`⟩ A simple specialization of *RuleGenerator* that can be used if the system is compiled without the RSES library. Functions as a drop-in replacement for the corresponding RSES wrapper.

2.3.4.34 ⟨`kernel/algorithms/loader.*`⟩ Specialization of *ScriptAlgorithm*. Invokes the input structure's *Load* method.

2.3.4.35 ⟨`kernel/algorithms/manualreducer.*`⟩ Specialization of *Reducer* that enables one to manually create a *Reducts* objects with a single *Reduct* member object.

2.3.4.36 ⟨`kernel/algorithms/manualscaler.*`⟩ Specialization of *Scaler* that enables one to manually discretize an attribute. Relevant to use if, e.g., suitable domain knowledge exists.

2.3.4.37 ⟨`kernel/algorithms/matlabdecisiontableexporter.*`⟩ Exports the contents of the input *DecisionTable* object to an ASCII file in a format recognized by MATLAB [10]. Enables tabular data to be visualized outside of ROSETTA.

2.3.4.38 ⟨`kernel/algorithms/meancompleter.*`⟩ Specialization of *Completer* where a missing value for an attribute is replaced by the mean value of all observed values for that attribute. If the attribute is non-numeric, the mode[10] is employed instead of the mean.

2.3.4.39 ⟨`kernel/algorithms/mydecisiontableexporter.*`⟩ Exports the contents of the input *DecisionTable* object to an ASCII file in a plain and simple format. The first two lines of the exported file contain attribute names and types, while each remaining line holds data for an object.

[10]The most frequently occurring value.

2.3.4.40 ⟨`kernel/algorithms/mydecisiontableimporter.*`⟩ Enables tabular data exported by *MyDecisionTableExporter* to be read back into ROSETTA. Supports a simple file format that can be prepared outside of ROSETTA for easy import of alien data.

2.3.4.41 ⟨`kernel/algorithms/myposdecisiontableimporter.*`⟩ Fills the input *DecisionTable* object with 0/1 entries, constructed on the basis of a definition of a Boolean POS function. See [12] for details.

TIP This enables ROSETTA to be used for more general Boolean reasoning purposes. By providing a gateway from function definitions to decision tables, ROSETTA can be used to compute prime implicants of any user-defined Boolean POS function. See also the description of the *Reducer* class.

The current implementation loads the full function definition before it is parsed. This approach does not scale up well to very large functions.

2.3.4.42 ⟨`kernel/algorithms/myreductexporter.*`⟩ Exports the contents of a *Reducts* object to an ASCII file in a plain and simple format.

2.3.4.43 ⟨`kernel/algorithms/myreductfilter.*`⟩ Specialization of *ReductFilter*, where individual *Reduct* member objects are filtered away from a *Reducts* object on the basis of such criteria as, e.g., length and support.

2.3.4.44 ⟨`kernel/algorithms/myreductimporter.*`⟩ Enables data exported by *MyReductExporter* to be read back into ROSETTA. Needs access to the *DecisionTable* object from which the reducts stem.

2.3.4.45 ⟨`kernel/algorithms/myreductshortener.*`⟩ Shortens each *Reduct* object in a *Reducts* object according to various criteria. By shortening is meant removing one or more attribute indices.

2.3.4.46 ⟨`kernel/algorithms/myruleexporter.*`⟩ Exports the contents of a *Rules* object to an ASCII file in a plain and simple format.

2.3.4.47 ⟨`kernel/algorithms/myrulefilter.*`⟩ Specialization of *RuleFilter*, where individual *Rule* member objects are filtered away from a *Rules* object on the basis of criteria such as, e.g., accuracy, coverage, length, and dominating decision value.

2.3.4.48 ⟨`kernel/algorithms/naivebayesclassifier.*`⟩ Specialization of *Classifier* that implements a naive Bayes classifier. Conditional and prior probabilities are estimated from a "master" *DecisionTable* object.

2.3.4.49 ⟨`kernel/algorithms/naivescaler.*`⟩ Specialization of *OrthogonalScaler*. A univariate, supervised discretization algorithm where a cut is added if the objects neighboring the cut belong to different decision classes.[11] Usually generates far too many cuts than needed. Used by, e.g., the *JohnsonReducer* class to generate all candidate cuts.

2.3.4.50 ⟨`kernel/algorithms/objectselector.*`⟩ Helper class useful in situations where a subset of objects have to be selected. In use by, e.g., the *Reducer* class.

[11] More or less. See *FindCuts* in ⟨`kernel/algorithms/naivescaler.*`⟩ for details.

2.3.4.51 ⟨`kernel/algorithms/objecttrackingvoter.*`⟩ Specialization of *Voter*. Tracks back to all objects that are members of the support set of a firing rule, and examines the distribution of decision vales among these objects.

2.3.4.52 ⟨`kernel/algorithms/orthogonalfilescaler.*`⟩ Specialization of *OrthogonalScaler*. Reads the cuts from a file instead of computing them.

2.3.4.53 ⟨`kernel/algorithms/orthogonalscaler.*`⟩ Base class for discretization algorithms that compute cuts. A cut can be perceived as a hyperplane orthogonal to the attribute axes, hence the name *OrthogonalScaler*. Symbolic attributes can be masked away before invoking the discretization process.

TIP If you want to write a univariate discretization algorithm, a lot of the functionality you'll need is already implemented at the *OrthogonalScaler* level. In the simplest case, all you'll need to do is to overload the *FindCuts* method. For more advanced discretization algorithms, e.g., multivariate algorithms, you might need to overload the *Discretize* method.

2.3.4.54 ⟨`kernel/algorithms/parallelexecutor.*`⟩ Specialization of *Executor* that, conceptually, executes different commands or command scripts in parallel. That is, the same input *Structure* is fed into all commands listed in the script.

2.3.4.55 ⟨`kernel/algorithms/partitioner.*`⟩ Partitions a decision table, i.e., computes and returns the sets of objects that are observationally equivalent wrt. a specified set of attributes. Can be seen as a wrapper around the functionality in the *PartitionKit* class.

2.3.4.56 ⟨`kernel/algorithms/prologdecisiontableexporter.*`⟩ Exports a *DecisionTable* object as a collection of Prolog facts.

2.3.4.57 ⟨`kernel/algorithms/prologreductexporter.*`⟩ Exports a *Reducts* object as a collection of Prolog facts.

2.3.4.58 ⟨`kernel/algorithms/prologruleexporter.*`⟩ Exports a *Rules* object as a collection of Prolog rules. The head of each exported rule contains various numerical information associated with the rule. Inconsistent rules are split into several individually consistent rules.

2.3.4.59 ⟨`kernel/algorithms/qualityrulefilter.*`⟩ Specialization of *RuleFilter*. Enables members of a *Rules* object to be filtered away according to the measures of rule quality listed by Bruha [3]. The quality measures are implemented by the classes in the *RuleEvaluator* family.

2.3.4.60 ⟨`kernel/algorithms/qualityrulefilterloop.*`⟩ Couples *QualityRuleFilter* together with a *Voter* algorithm and ROC analysis. Enables the classificatory performance of the set of rules to be monitored as a function of the quality threshold.

2.3.4.61 ⟨`kernel/algorithms/reducer.*`⟩ Base class for algorithms that compute reducts of information systems, modulo the decision attribute or not. Reducts may be computed wrt. the full system or relative to selected objects. Support is provided for IDGs and boundary region thinning. The returned *Reducts* object may or may not have a child *Rules* object attached to it.

TIP Since a reduct is a prime implicant of a suitably constructed Boolean POS function, we can employ algorithms in the *Reducer* family for more general Boolean reasoning purposes. To compute the prime implicants of a problem-specific POS function, invoke the overloaded *ComputePrimeImplicants* method.

TIP Algorithms in the *Reducer* family expect that the input *DecisionTable* object is already discretized, if needed. Some newly proposed algorithms for computing reducts and rules, however, do the discretization and the reduction during the same pass. Such algorithms could be incorporated by creating a new algorithm family, e.g., *ScalingReducer*, which return the discretized version of the input *DecisionTable* object, but with a *Reducts* object attached as a child.

2.3.4.62 ⟨`kernel/algorithms/reductcostfilter.*`⟩ Specialization of *ReductFilter*, where individual reducts are filtered away if they "cost" too much. Cost evaluation is done by the *CostInformation* helper class.

2.3.4.63 ⟨`kernel/algorithms/reductexporter.*`⟩ Base class for algorithms that export *Reducts* objects to some alien format.

2.3.4.64 ⟨`kernel/algorithms/reductfilter.*`⟩ Specialization of *Filter* that operates on *Reducts* objects.

2.3.4.65 ⟨`kernel/algorithms/reductimporter.*`⟩ Base class for algorithms that import sets of reducts.

2.3.4.66 ⟨`kernel/algorithms/reductperformancefilter.*`⟩ Specialization of *ReductFilter*. Each member *Reduct* object is evaluated according to the classificatory performance of the rules generated from that *Reduct* object alone.

2.3.4.67 ⟨`kernel/algorithms/reductshortener.*`⟩ Base class for algorithms that remove attributes from *Reduct* objects according to some criterion.

2.3.4.68 ⟨`kernel/algorithms/removalcompleter.*`⟩ Specialization of *Completer*. Removes all rows that contain one or more missing values.

2.3.4.69 ⟨`kernel/algorithms/reporter.*`⟩ Base class for algorithms that generate a report or log of the current experiment. Typically, annotations and other information belonging to the input *Structure* object are saved, and the same is then done recursively on the input object's children.

2.3.4.70 ⟨`kernel/algorithms/rulebasedclassifier.*`⟩ Specialization of *Classifier*. For algorithms that employ a *Rules* object together with some conflict resolution scheme to construct and return a *RuleBasedClassification* object.

2.3.4.71 ⟨`kernel/algorithms/ruleevaluator.*`⟩ The *RuleEvaluator* family implements rule quality measures [3], i.e., they evaluate how "good" a *Rule* object is in some sense. Used by, e.g., the *QualityRuleFilter* algorithm.

2.3.4.72 ⟨`kernel/algorithms/ruleexporter.*`⟩ Base class for algorithms that export a *Rules* object to some alien format.

2.3.4.73 ⟨`kernel/algorithms/rulefilter.*`⟩ Abstract subclass of *Filter*, specialized for removing *Rule* objects from a *Rules* object according to some filtering criterion.

2.3.4.74 ⟨`kernel/algorithms/rulegenerator.*`⟩ Base class for algorithms that produce a *Rules* object from a *Reducts* object. Typically, if the input *Reducts* object already has a *Rules* object as a child, this is returned. Otherwise, the *Rules* object is generated by laying the individual *Reduct* objects over their ancestral *DecisionTable* object and reading off the corresponding table entries.

2.3.4.75 ⟨`kernel/algorithms/saver.*`⟩ Specialization of *ScriptAlgorithm*. Invokes the input structure's *Save* method.

2.3.4.76 ⟨`kernel/algorithms/scaler.*`⟩ Base class for algorithms that perform discretization, i.e., that somehow recode the attributes in a *DecisionTable* object to provide a coarser view of the world. The name *Scaler* is kept for historical reasons.

2.3.4.77 ⟨`kernel/algorithms/scriptalgorithm.*`⟩ Base class for algorithms that are only intended used in command scripts. Command scripts are executed by algorithms in the *Executor* family.

2.3.4.78 ⟨`kernel/algorithms/seminaivescaler.*`⟩ Specialization of *NaiveScaler* which produces slightly fewer cuts. The sets of dominating decisions around a candidate cut are examined, and the cut is omitted if these sets define an inclusion.

2.3.4.79 ⟨`kernel/algorithms/serialexecutor.*`⟩ Specialization of *Executor* where the command script is interpreted as defining a pipeline, i.e., where data flows from one algorithm to the next.

2.3.4.80 ⟨`kernel/algorithms/serialexecutorloop.*`⟩ Specialization of *SerialExecutor* where the command script is interpreted as defining both a training pipeline and a testing pipeline, that are to be executed multiple times on different resampled versions of the input *DecisionTable* object.

TIP The resampling scheme can be overloaded by subclasses of *SerialExecutorLoop* to provide for, e.g., cross-validation or bootstrapping.

2.3.4.81 ⟨`kernel/algorithms/splitter.*`⟩ Base class for algorithms that take as input a *DecisionTable* object, and splits this horizontally into two or more subtables according to some splitting criterion.

2.3.4.82 ⟨`kernel/algorithms/standardvoter.*`⟩ Implements a rule-based classification scheme where conflict resolution among firing rules is resolved through traditional voting. A firing rule gets to cast a certain number of votes in favour of the decision values it indicates. All votes are tallied and their relative percentages returned.

2.3.4.83 ⟨`kernel/algorithms/structurecreator.*`⟩ Specialization of *ScriptAlgorithm*. Ignores the input *Structure*, and outputs the result of a call to the static *Creator* helper class.

2.3.4.84 ⟨`kernel/algorithms/valuesplitter.*`⟩ Specialization of *Splitter*. All objects in each resulting subtable will have the same value for some specified attribute.

2.3.4.85 ⟨`kernel/algorithms/voter.*`⟩ Classifies a possibly incomplete *InformationVector* object according to some rule-based voting scheme.

2.3.5 ⟨`kernel/utilities`⟩

Contains code for various utilities, such as random number generators, tools for statistical hypothesis testing, computation of partitions, common mathematical operations, etc.

2.3.5.1 ⟨`kernel/utilities/binaryoutcomecomparator.*`⟩ Base class for doing statistical hypothesis testing on classifiers with binary outcomes.

2.3.5.2 ⟨`kernel/utilities/cindexcomputer.*`⟩ Utility class that computes the area under the ROC curve and its standard error non-parametrically [6]. Used by, e.g., the *ROCCurve* class.

2.3.5.3 ⟨`kernel/utilities/creator.*`⟩ Static kit for dynamic creation of structural objects. Provides support for creating objects by loading them from disk, too.

TIP Never use *new* to create structural objects, use the static *Creator* class instead.[12] The *Creator* class enables us to create objects while working on the level of abstract base classes.

The *Creator* class works by scanning the static *ObjectManager* class for installed prototype objects, and then cloning the most closely matching[13] structural object by invoking its *Duplicate* method.

2.3.5.4 ⟨`kernel/utilities/delimiter.*`⟩ Scope around delimiter constants. Used by, e.g., the *IOKit* class.

2.3.5.5 ⟨`kernel/utilities/discerner.*`⟩ Helper class for algorithms that need a discernibility predicate on a per attribute basis. The *Discerner* class provides support for IDGs.

2.3.5.6 ⟨`kernel/utilities/hanleymcneilcomparator.*`⟩ Specialization of *BinaryOutcomeComparator* implementing Hanley-McNeil's test [7]. Used to compare areas under ROC curves derived from the same set of cases.

2.3.5.7 ⟨`kernel/utilities/iokit.*`⟩ Provides a set of static methods for I/O related functions, e.g., loading a line from a stream while ignoring comment lines, or saving a quoted string.

2.3.5.8 ⟨`kernel/utilities/mathkit.*`⟩ Provides a scope around miscellaneous mathematical and statistical methods, e.g., for computing histograms, mean and median values, variances, correlations, linear regressions, etc.

2.3.5.9 ⟨`kernel/utilities/mcnemarcomparator.*`⟩ Specialization of *BinaryOutcomeComparator* implementing McNemar's test. Used to compare classification accuracies for classifiers applied to the same set of cases.

[12]Never use *delete* either, use the *Handle* mechanism instead.
[13]Using the *GetId* and *IsA* methods inherited from *Identifier*.

2.3.5.10 ⟨`kernel/utilities/partitionkit.*`⟩ Utility functions for computing the equivalence classes in a decision table wrt. a given set of attributes. Implemented by a simple but efficient sort-and-scan procedure.

2.3.5.11 ⟨`kernel/utilities/permuter.*`⟩ Helper class useful in situations where we need to create a permutation of the index set $\{0, \ldots, n-1\}$, such that the permutation is sorted according to a given vector of sorting keys.

2.3.5.12 ⟨`kernel/utilities/rng.*`⟩ Implements a good random number generator for uniform distributions, lifted from Press et al. [14]. Both reals and integers can be drawn.

2.3.5.13 ⟨`kernel/utilities/systemkit.*`⟩ Provides various helper functions for obtaining timestamps, user names, etc.

2.3.6 ⟨`kernel/rses`⟩

Contains code relevant to the RSES library.

2.3.6.1 ⟨`kernel/rses/rsesmessage.*`⟩ Implements a global method used by the RSES library.[14]

TIP The RSES library assumes that the global methods listed below are available. The current implementation of these methods use the facilities provided by the *Message* class.

> *void Message(char *, char *)*
> *int stop(int, ...)*

2.3.6.2 ⟨`kernel/rses/rsesstop.*`⟩ Implements a global method used by the RSES library.[15]

2.3.6.3 ⟨`kernel/rses/library`⟩ Contains legacy code from the RSES library.[16]

2.3.6.4 ⟨`kernel/rses/structures`⟩ Contains encapsulating wrappers for some of the structural objects from the RSES library.

2.3.6.4.1 ⟨`kernel/rses/structures/rsesdecisiontable.*`⟩ Provides a wrapper around the RSES class *TDTable*.

2.3.6.4.2 ⟨`kernel/rses/structures/rsesinformationvector.*`⟩ Provides a wrapper around the RSES class *TDObject*.

2.3.6.4.3 ⟨`kernel/rses/structures/rsesreduct.*`⟩ Provides a wrapper around the RSES class *TReduct*.

2.3.6.4.4 ⟨`kernel/rses/structures/rsesreducts.*`⟩ Provides a wrapper around the RSES class *TRedRulMem*. A lot of administration goes into ensuring consistency between the embedded *TReduct* objects and the encapsulating *RSESReduct* wrappers.

[14]See ⟨`kernel/rses/rsesmessage.*`⟩ for details.
[15]See ⟨`kernel/rses/rsesstop.*`⟩ for details.
[16]Developed at the Group of Logic, University of Warsaw, Poland.

2.3.6.4.5 ⟨`kernel/rses/structures/rsesrule.*`⟩ Provides a wrapper around the RSES class *TRule*. Also keeps a pointer to the RSES *TReduct* object to which the *TRule* object was derived.

2.3.6.4.6 ⟨`kernel/rses/structures/rsesrules.*`⟩ RSES bundles all reducts and rules into the compound RSES class *TRedRulMem*. To still be able to achieve a logical separation between sets of reducts and sets of rules, the *RSESRules* wrapper keeps a pointer to an RSES *TRedRulMem* object and does a lot of housekeeping to ensure consistency.

2.3.6.5 ⟨`kernel/rses/algorithms`⟩ Contains encapsulating wrappers for some of the algorithms from the RSES library.

2.3.6.5.1 ⟨`kernel/rses/algorithms/rsesclassifier.*`⟩ Provides a wrapper around the RSES class *TDecGenerator*.

2.3.6.5.2 ⟨`kernel/rses/algorithms/rsesdecisiontableimporter.*`⟩ Provides a wrapper around the RSES *TDictionary::TDimport* method.

2.3.6.5.3 ⟨`kernel/rses/algorithms/rsesdynamicreducer.*`⟩ Provides a wrapper around the methods from the RSES library that deal with computing dynamic reducts as proposed by Bazan et al. [2]. The *RSESStaticReducer* object given as a parameter basically does all the work. The *RSESDynamicReducer* invokes suitable RSES methods prior to this.

2.3.6.5.4 ⟨`kernel/rses/algorithms/rsesexhaustivereducer.*`⟩ Specialization of *RSESStaticReducer*. Computes all reducts in an exhaustive manner. Suitable for small systems only.

2.3.6.5.5 ⟨`kernel/rses/algorithms/rsesgeneticreducer.*`⟩ Specialization of *RSESStaticReducer*. Provides a wrapper around the RSES implementation of the genetic algorithm of Wróblewski [18].

2.3.6.5.6 ⟨`kernel/rses/algorithms/rsesjohnsonreducer.*`⟩ Specialization of *RSESStaticReducer*. Provides a wrapper around the RSES implementation of the greedy algorithm of Johnson [9]. See the *JohnsonReducer* class for an alternative implementation.

2.3.6.5.7 ⟨`kernel/rses/algorithms/rsesorthogonalfilescaler.*`⟩ Specialization of *OrthogonalFileScaler*. Provides a wrapper around the implementation offered by the RSES library.

2.3.6.5.8 ⟨`kernel/rses/algorithms/rsesorthogonalscaler.*`⟩ Provides a wrapper around the RSES implementation of the discretization algorithm of Nguyen and Skowron [11]. A very fast and efficient implementation, but with no support for approximate solutions. See also the description of the *BROrthogonalScaler* class.

2.3.6.5.9 ⟨`kernel/rses/algorithms/rsesreducer.*`⟩ Specialization of *Reducer* for algorithms hailing from the RSES library. No support provided for IDGs or boundary region thinning.

2.3.6.5.10 ⟨`kernel/rses/algorithms/rsesrulegenerator.*`⟩ Provides a wrapper around the RSES *TRedRulCalc::GenerateRules* method. No IDG support provided.

2.3.6.5.11 ⟨`kernel/rses/algorithms/rsesrulelessreductfilter.*`⟩ Specialization of *ReductFilter*. Removes a *RSESReduct* object if the embedded RSES *TReduct* object has no *TRule* object derived from it.

2.3.6.5.12 ⟨`kernel/rses/algorithms/rsesstaticreducer.*`⟩ Base class for algorithms from the RSES library that compute proper reducts, and that can be used as a subcomponent of the *RSESDynamicReducer* class. Provides a wrapper around methods in the RSES *TRedRulCalc* class. Algorithms in the *RSESStaticReducer* family do not provide support for approximate reducts.

2.3.7 ⟨`kernel/sav`⟩

Contains code relevant to the SAV library.

2.3.7.1 ⟨`kernel/sav/library`⟩ Contains legacy code from a library for genetic computations.[17]

2.3.7.1.1 ⟨`kernel/sav/library/ea`⟩ Contains general code for genetic algorithms.

2.3.7.1.2 ⟨`kernel/sav/library/hits`⟩ Contains code specific for applying genetic algorithms to compute minimal hitting sets.

2.3.7.2 ⟨`kernel/sav/algorithms`⟩ Contains encapsulating wrappers for some of the algorithms from the SAV library.

2.3.7.2.1 ⟨`kernel/sav/algorithms/savgeneticreducer.*`⟩ Provides a wrapper around methods in the SAV *Hits* class for computing minimal hitting sets. Implements the algorithm of Vinterbo and Øhrn [17]. Support for approximate solutions is provided.

2.4 ⟨`common`⟩

Contains potentially front-end dependent code called from the kernel, e.g., methods for giving error messages, installing prototype objects and linking them to their associated dialogs, etc.

2.4.1 ⟨`common/configuration.*`⟩

Static scope around system configuration variables.

TIP Make sure this file is compiled as part of any released builds. That way, the build timestamp is sure to be valid.

2.4.2 ⟨`common/installer.*`⟩

Determines which algorithms and structures that are available to the user, i.e., that are "installed" as prototype objects in the *ObjectManager* pool.

TIP Use the *INSTALLSTRUCTURE* and *INSTALLALGORITHM* macros to install prototype structures and algorithms. For a class to be available throughout the system, a prototype object has to be installed.

The *INSTALLALGORITHM* macro also associates an *Algorithm* object with its GUI dialog box, if relevant.

[17] Developed by Staal A. Vinterbo, Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway.

2.4.3 ⟨`common/messagehelper.*`⟩

Static helper class for the *Message* class. Takes care of routing the messages to the GUI or wherever.

TIP Methods from the *MessageHelper* class should never be called directly from the kernel. Use the *Message* class instead.

2.4.4 ⟨`common/objectmanager.*`⟩

Manages the algorithms and structures that are available to the user, i.e., that are "installed" as prototype objects. Objects are installed by the static *Installer* class. Used by, e.g., the static *Creator* class.

2.5 ⟨`frontend`⟩

Contains the GUI front-end of ROSETTA.

2.5.1 ⟨`frontend/algorithms`⟩

Contains algorithmic objects that depend on MFC and the Windows platform, specifically ODBC code.

2.5.2 ⟨`frontend/dialogs`⟩

Contains code for handling the logic behind the dialog boxes.

2.5.2.1 ⟨`frontend/dialogs/structuredialogs`⟩ Contains code for handling the logic behind the dialog boxes related to structural objects such as statistics dialogs, etc.

2.5.2.2 ⟨`frontend/dialogs/algorithmdialogs`⟩ Contains code for handling the logic behind the dialog boxes related to algorithmic objects such as dialogs for entering algorithm parameters, etc.

2.5.3 ⟨`frontend/views`⟩

Contains code for displaying structural objects in the front-end such as views for decision tables, project trees, etc.

3 Compiler Settings

In order to compile and make use of the ROSETTA C++ library, you need to create a *make* file. If you are using Microsoft Visual C++, you will probably use the workspace feature of the development environment to do this, thus hiding away the details of the *make* (or *nmake*) file.

For examples of *make* or workspace files, see [15].

3.1 Directory Search Order

The ROSETTA C++ library includes files in brackets. A library like STL, however, includes files in double quotes. This might not cause any problems for you, but if it does you can get around it by redefining the order in which include directories are searched. You will have to specify the path that the bracket-included files are relative to, anyway.

Let `$ROSETTA` denote the directory where you have installed the ROSETTA C++ library, e.g., `/source/rosetta`. Then, let the list of directories searched start with the following:[18]

```
$ROSETTA/kernel/system/stl
$ROSETTA
```

3.2 Preprocessor Flags

There are several preprocessor flags worth knowing about:

_OLDCASTS Set this flag if your compiler is so old that it does not have support for the new C++-style casting mechanisms. If not set, the constructs *const_cast*, *dynamic_cast*, *static_cast* and *reinterpret_cast* will be used.

_MEMTMPL Set this flag if your compiler supports member templates in a robust manner. (For example, Microsoft Visual C++ 6.0 is not robust enough.) This enables the *Handle* template to be used in a more flexible manner.[19]

_RSES Set this flag if you are compiling with the RSES library. If set, structures and algorithms from the RSES library will be installed by the *Installer* class. Otherwise, simple drop-in substitute classes will be installed instead.

_DEBUG Set this flag if you are compiling a debug version of the library. If set, several methods will perform range checking, output debug messages, and do more extensive validation of parameters and data. As a consequence, the library will run in a slightly more sluggish manner if this flag is set.

_ROSGUI Set this flag if you are compiling with the ROSETTA GUI.

_MSGGUI Set this flag if you want the *MessageHelper* class to use MFC message methods, even though you are not compiling with the ROSETTA GUI. If not set, messages are redirected to *cout* and *cerr* instead.

_MSC_VER This flag is automatically set if you are using Microsoft Visual C++, and then holds the compiler's version number.

WIN32 Set this flag if you are compiling under 32-bit Windows. (If you are using Microsoft Visual C++, this is done automatically.) If set, some constants in *Bits* are hardcoded, thus avoiding one multiplication.

STLport [16], the STL implementation that accompanies the ROSETTA C++ library, has a wealth of flags that can be set. For most compilers and platforms, you won't have to dabble with these as STLport is able to autodetect sensible settings. However, you can set them explicitly, if necessary:

__STL_NO_NEW_IOSTREAMS The ROSETTA C++ library does not make use of the new iostream run-time libraries. Set this flag to signal that STLport should not either.

If you are using Microsoft Visual C++, you might want to know about the following preprocessor flags:

_AFX_NO_DEBUG_CRT In debug mode, Microsoft Visual C++ has its own version of *new*. If this causes problems for the *Referent* overloadings of *new*, try defining this flag.

[18]Under Microsoft Visual C++ 6.0, you specify this under the *Directories* tab in the dialog box that appears if you select the *Tools/Options...* menu entry.

[19]For reasons of backwards compatibility and lack of member template support, the ROSETTA C++ library in several places uses ordinary pointers instead of *Handle* objects in method interfaces. This in turn makes it necessary to make slightly awkward calls to *Handle::GetPointer* to pass parameters across these interfaces.

4 Miscellaneous

4.1 Exceptions

For historical reasons, mainly, the ROSETTA C++ library does not really make use of the exception handling mechanisms of C++, i.e., *try*, *catch* and *throw*. But doing so would have made error handling simpler in some places. Instead, the ROSETTA C++ library has adopted the convention of checking the return values of methods. Methods generally return *false*, *NULL* or an undefined value[20] if they fail.

4.2 Coding Standards

Coding standards, i.e., conventions and rules regarding syntax, help make code more perspicuous, streamlined and easier to read. Some conventions in the ROSETTA C++ library are worth knowing about, and are listed below.[21] (It is probably easier to interpret the items below if you study an example file at this time, e.g., ⟨`kernel/structures/graph.*`⟩.)

- An indentation unit of 2 is used throughout. Set the tab properties of your editor accordingly.
- Names of classes and methods are capitalized. Uppercase letters midword are used for names that logically span more than one word.
- Variable names are written in lowercase. Underscores midword may be used for names that logically span more than one word.
- Member variables always have a trailing underscore in their names.
- Method implementations have a standardized header. Files and class definitions carry a similar standardized header.
- Declarations of member variables and member methods are vertically aligned in the class definitions.
- Declarations of overloaded methods are prefixed with a header line stating at which superclass the methods were originally defined. The order these are given in should correspond with the class hierarchy. Constructors are listed first, new or local methods are listed last.
- Most macros are written in uppercase letters. Container class macros and casting macros are exceptions to this rule.
- A single space is always added between such keywords as, e.g., *if* and *for*, their associated parentheses, and curly braces. Curly braces occur on the same line as the keyword.

  ```
  if (i > 0) {
    for (j = 0; j < i; j++) {
      DoThis(i, i + j);
      DoThat(i, i - j);
    }
  }
  else {
    DoThis(-i, -i);
  }
  ```

  Similarly, a single space is added after commas, around binary operators, etc.
- More instances than necessary of the keywords *public*, *protected* and *private* may be used to separate between declarations of member variables and member methods.
- In method implementations, the return type is given on a separate line above the method head.

[20] See the description of the static *Undefined* class in ⟨`kernel/basic/undefined.*`⟩.

[21] There are probably several more items that should be added to the list but that are not. Either because I have forgotten to list them, or, most likely, because I am too lazy to complete the list.

A Algorithm Signatures

In general, classes derived from *Algorithm* and *Structure* interact via the *Apply* method. Algorithmic objects take as input structural objects, and return structural objects. The output object may be the same as the input object, or it may be a new object. Type signatures for classes derived from *Algorithm* can be found in Table 1.

Class	Input type	Output type
Approximator	*DecisionTable*	*Approximation*
BatchClassifier	*DecisionTable*	*BatchClassification*
Classifier	*InformationVector*	*Classification*
Completer	*DecisionTable*	*DecisionTable*
DecisionTableExporter	*DecisionTable*	*DecisionTable*
DecisionTableImporter	*DecisionTable*	*DecisionTable*
DictionaryExporter	{*Dictionary*, *DecisionTable*}	{*Dictionary*, *DecisionTable*}
DictionaryImporter	{*Dictionary*, *DecisionTable*}	{*Dictionary*, *DecisionTable*}
Executor	*Structure*	*Structure*
Partitioner	*DecisionTable*	*EquivalenceClasses*
ReductExporter	*Reducts*	*Reducts*
ReductFilter	*Reducts*	*Reducts*
ReductImporter	*DecisionTable*	*Reducts*
Reducer	*DecisionTable*	*Reducts*
Reporter	*Structure*	*Structure*
RSESDecisionTableImporter	*RSESDecisionTable*	*RSESDecisionTable*
RSESOrthogonalFileScaler	*RSESDecisionTable*	*RSESDecisionTable*
RSESOrthogonalScaler	*RSESDecisionTable*	*RSESDecisionTable*
RSESReducer	*RSESDecisionTable*	*RSESReducts*
RSESRuleGenerator	*RSESReducts*	*RSESRules*
RSESRulelessReductFilter	*RSESReducts*	*RSESReducts*
RuleBasedClassifier	*InformationVector*	*RuleBasedClassification*
RuleExporter	*Rules*	*Rules*
RuleFilter	*Rules*	*Rules*
RuleGenerator	*Reducts*	*Rules*
SerialExecutorLoop	*DecisionTable*	*Structure*
Scaler	*DecisionTable*	*DecisionTable*
ScriptAlgorithm	*Structure*	*Structure*
Splitter	*DecisionTable*	{*DecisionTable*, *DecisionTables*}

Table 1: Type signatures for classes derived from *Algorithm*, i.e., input/output type relationships for the *Apply* method. For classes that due to inheritance match more than type, the most specific type listed applies.

B Class Hierarchies

The following figures display the class hierarchies in the ROSETTA C++ library. Only classes involved in inheritance are displayed, i.e., stand-alone classes are not shown. Classes from external libraries are also not shown. All inheritance is public, unless otherwise indicated.

Figure 1 displays the top level hierarchy. All classes that are derived from *Referent* can be used together with the *Handle* mechanism.

Figure 2 displays the first part of the *Structure* hierarchy. Classes derived from *Structure* are able to programmatically interface with classes derived from *Algorithm*. Not all classes derived from *Structure* do so, though, or are even intended to do so. Figure 2 is a subhierarchy of Figure 1.

Figure 3 displays the second part of the *Structure* hierarchy, and is a continuation of Figure 2.

Figure 4 displays the third part of the *Structure* hierarchy, and is a continuation of Figure 2 and Figure 3.

Figure 5 displays the *AnnotatedStructure* hierarchy. Structural objects that have *Annotation* objects associated with them are derived from this class. An object is typically annotated if a user wants to interact with it through some user interface. Figure 5 is a subhierarchy of Figure 2.

Figure 6 displays the first part of the *Algorithm* hierarchy. Classes derived from *Algorithm* are able to programmatically interface with classes derived from *Structure*. Figure 2 is a subhierarchy of Figure 1.

Figure 7 displays the second part of the *Algorithm* hierarchy, and is a continuation of Figure 6.

Figure 8 displays the *Classifier* hierarchy. An algorithm in this family classifies a single object, given its corresponding *InformationVector* structure, and returns a *Classification* structure that holds a set of possible classes. A *Classifier* object is typically used as a subcomponent of an algorithm in the *BatchClassifier* family. Figure 8 is a subhierarchy of Figure 6.

Figure 9 displays the *Completer* hierarchy. An algorithm in this family takes as input a *DecisionTable* object with missing values, and returns a *DecisionTable* object that has no missing values. Figure 9 is a subhierarchy of Figure 6.

Figure 10 displays the *Executor* hierarchy. Algorithms in this family execute command scripts. Figure 10 is a subhierarchy of Figure 6.

Figure 11 displays the *Exporter* hierarchy. Algorithms that export some kind of *Structure* object to some external medium in some format inherit from *Exporter*. Figure 11 is a subhierarchy of Figure 6.

Figure 12 displays the *DecisionTableExporter* hierarchy. Algorithms that export some aspect of a *DecisionTable* object to some format inherit from this class. Figure 12 is a subhierarchy of Figure 11.

Figure 13 displays the *Filter* hierarchy. Algorithms in *Filter* family remove individual structural objects from collections of such according to some filtering criterion. Figure 13 is a subhierarchy of Figure 7.

Figure 14 displays the *Importer* hierarchy. Algorithms that import structural objects from some medium inherit from *Importer*. Figure 14 is a subhierarchy of Figure 7.

Figure 15 displays the *Reducer* hierarchy. Algorithms in this family compute reducts of the input *DecisionTable* object. The returned *Reducts* structure may possibly have a *Rules* structure associated with it. Figure 15 is a subhierarchy of Figure 7.

Figure 16 displays the *RuleGenerator* hierarchy. Algorithms in this family produce *Rules* objects from *Reducts* objects. Figure 16 is a subhierarchy of Figure 7.

Figure 17 displays the *Scaler* hierarchy. Algorithms that inherit from *Scaler* discretize attributes in *DecisionTable* objects. Figure 17 is a subhierarchy of Figure 7.

Figure 18 displays the *ScriptAlgorithm* hierarchy. Algorithms should inherit from *ScriptAlgorithm* if they are only intended used in command scripts. Figure 18 is a subhierarchy of Figure 7.

Figure 19 displays the *Splitter* hierarchy. Algorithms in this family split *DecisionTable* objects into several distinct *DecisionTable* objects. Figure 19 is a subhierarchy of Figure 7.

Figure 20 displays miscellaneous classes where inheritance is involved.

Figure 21 displays the *RuleEvaluator* hierarchy. Classes in this hierarchy are helper classes for evaluating the "quality" of *Rule* objects. Figure 21 is a subhierarchy of Figure 1.

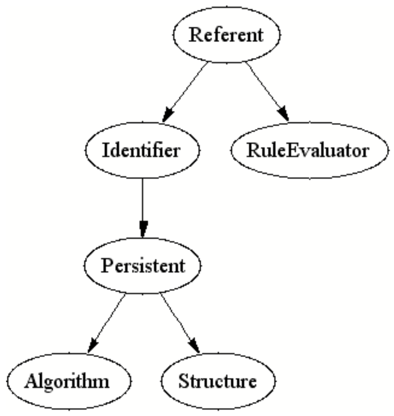

Figure 1: *Referent* hierarchy. Top level hierarchy.

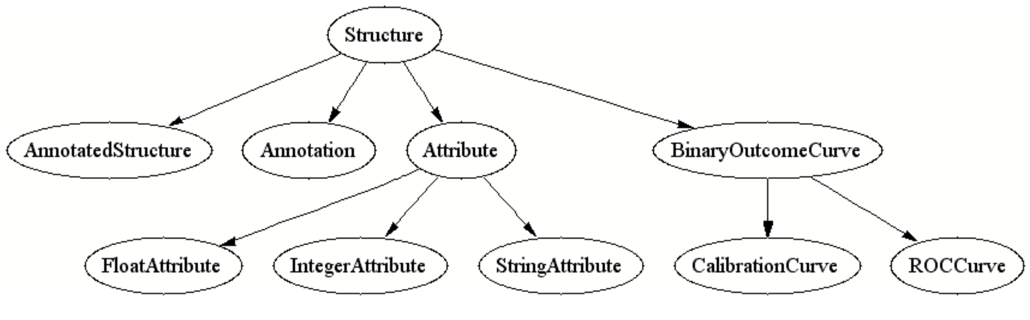

Figure 2: *Structure* hierarchy, part 1. Subhierarchy of Figure 1.

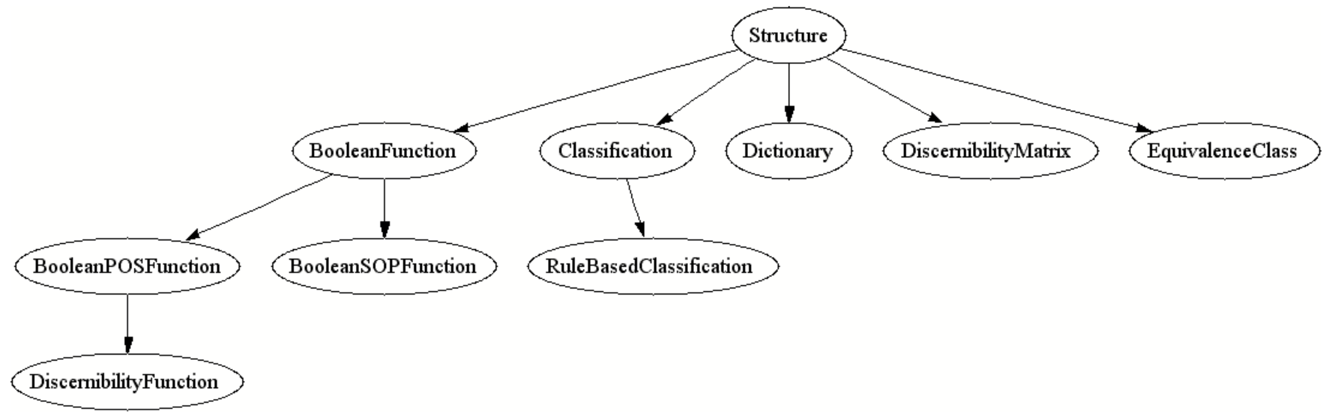

Figure 3: *Structure* hierarchy, part 2. Subhierarchy of Figure 1.

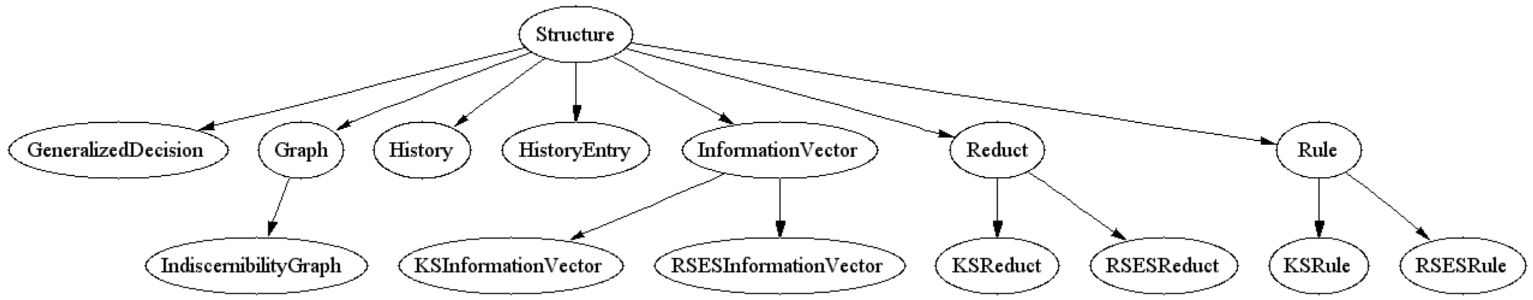

Figure 4: *Structure* hierarchy, part 3. Subhierarchy of Figure 1.

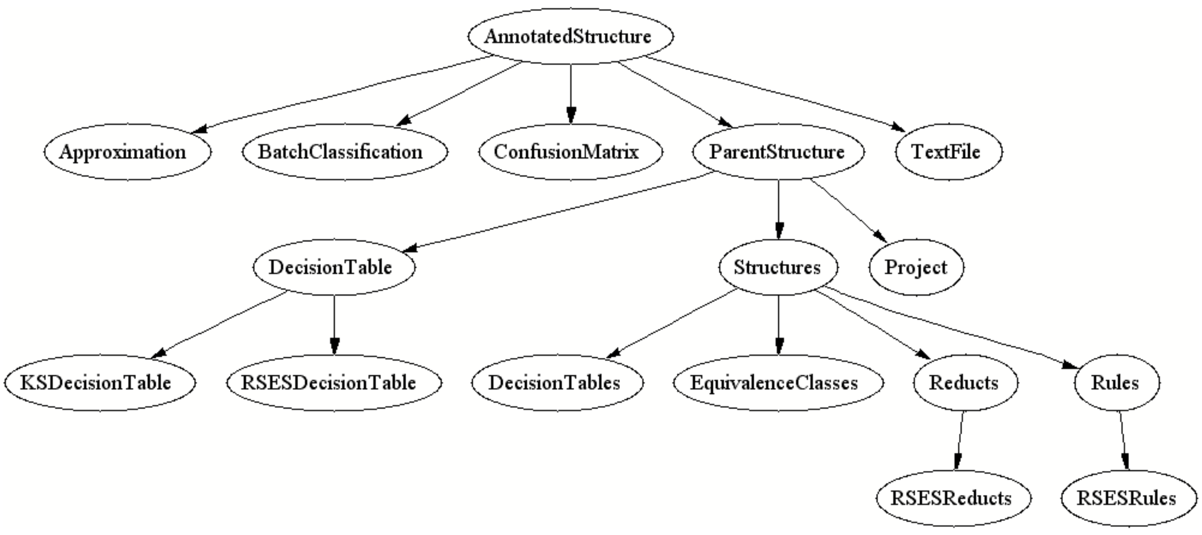

Figure 5: *AnnotatedStructure* hierarchy. Subhierarchy of Figure 2.

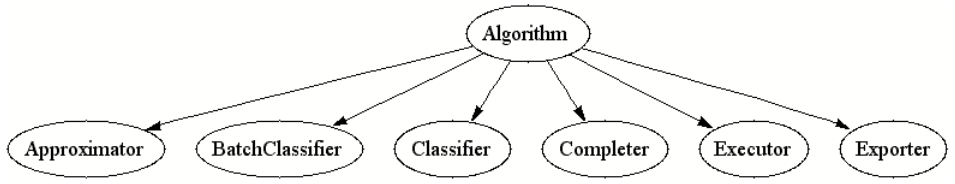

Figure 6: *Algorithm* hierarchy, part 1. Subhierarchy of Figure 1.

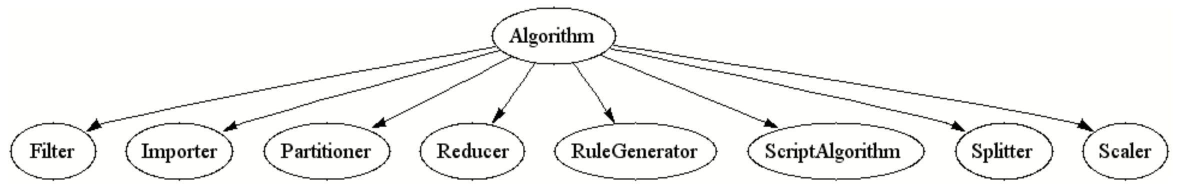

Figure 7: *Algorithm* hierarchy, part 2. Subhierarchy of Figure 1.

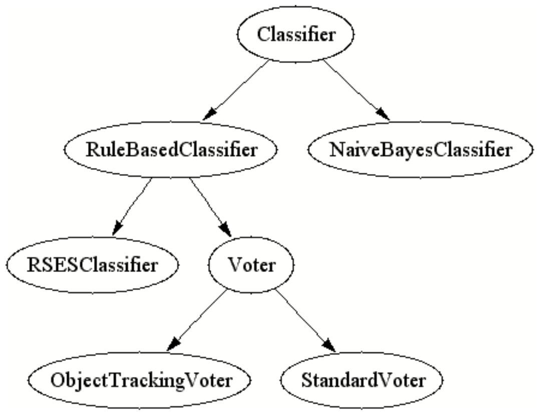

Figure 8: *Classifier* hierarchy. Subhierarchy of Figure 6.

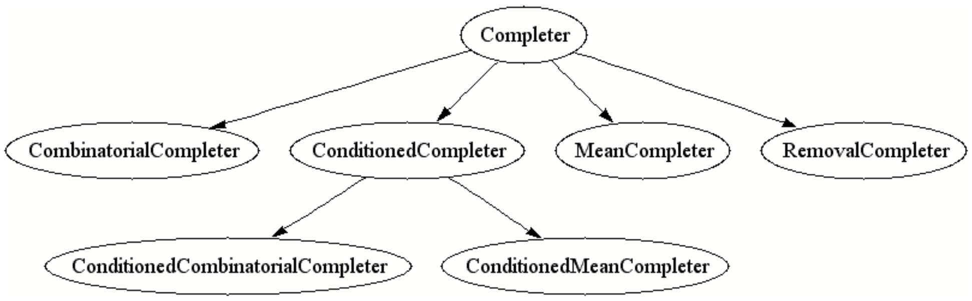

Figure 9: *Completer* hierarchy. Subhierarchy of Figure 6.

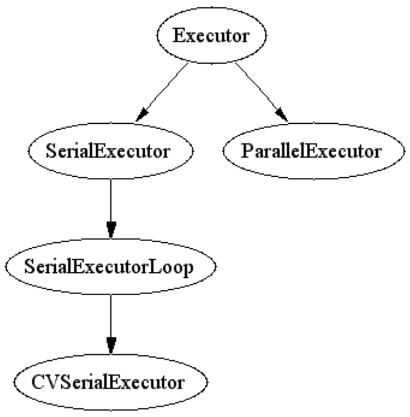

Figure 10: *Executor* hierarchy. Subhierarchy of Figure 6.

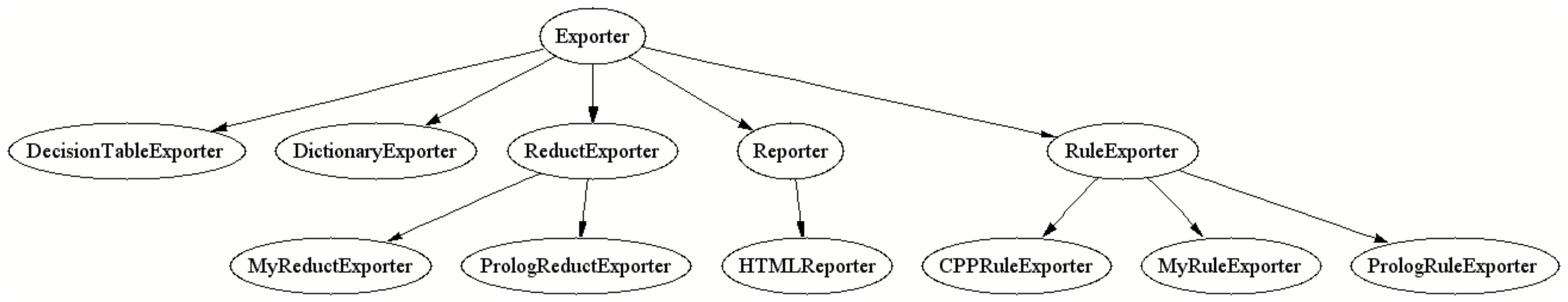

Figure 11: *Exporter* hierarchy. Subhierarchy of Figure 6.

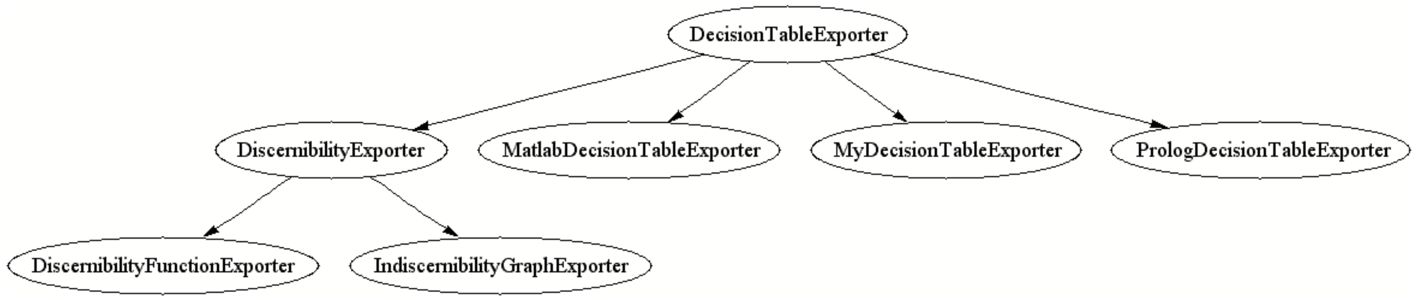
DecisionTableExporter
DiscernibilityExporter
MatlabDecisionTableExporter
MyDecisionTableExporter
PrologDecisionTableExporter
DiscernibilityFunctionExporter
IndiscernibilityGraphExporter

Figure 12: *DecisionTableExporter* hierarchy. Subhierarchy of Figure 11.

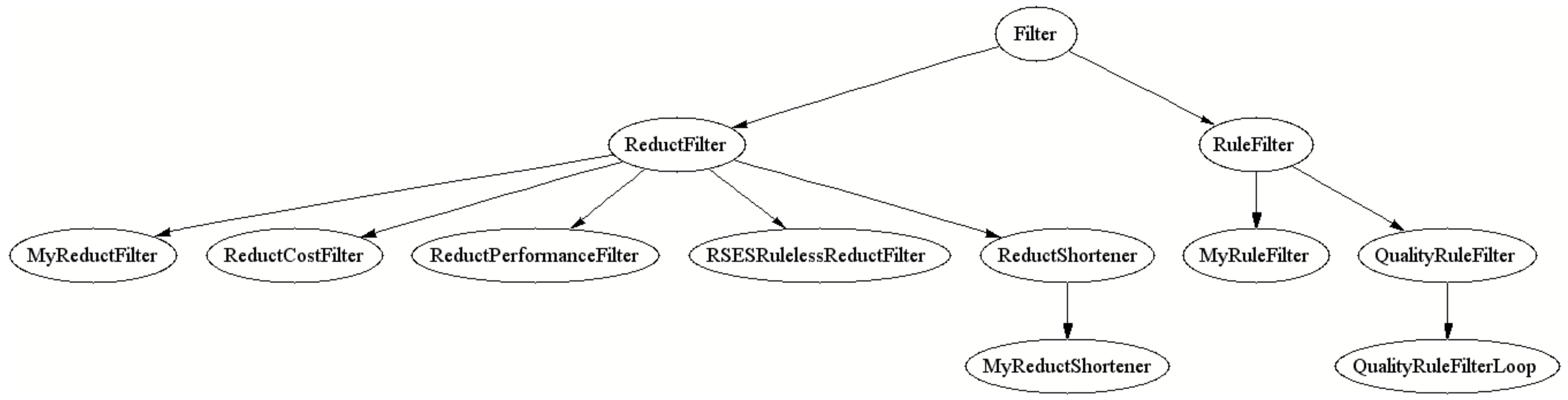

Figure 13: *Filter* hierarchy. Subhierarchy of Figure 7.

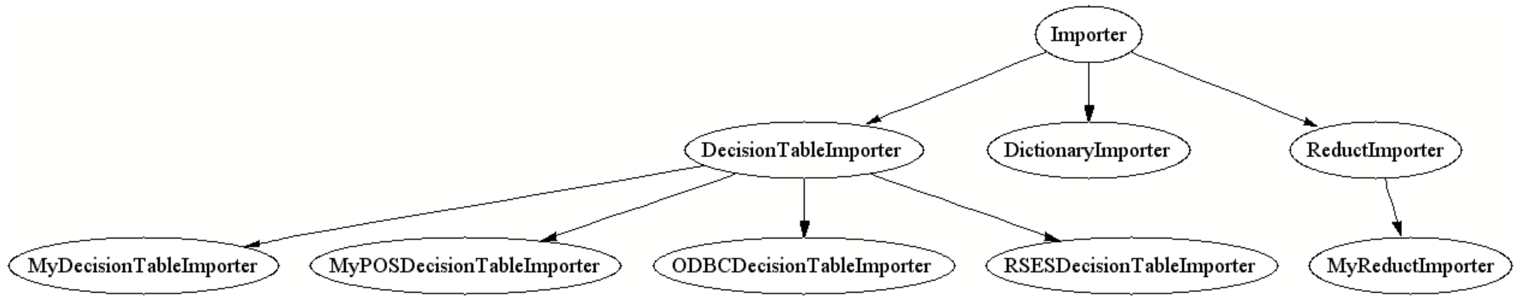

Figure 14: *Importer* hierarchy. Subhierarchy of Figure 7.

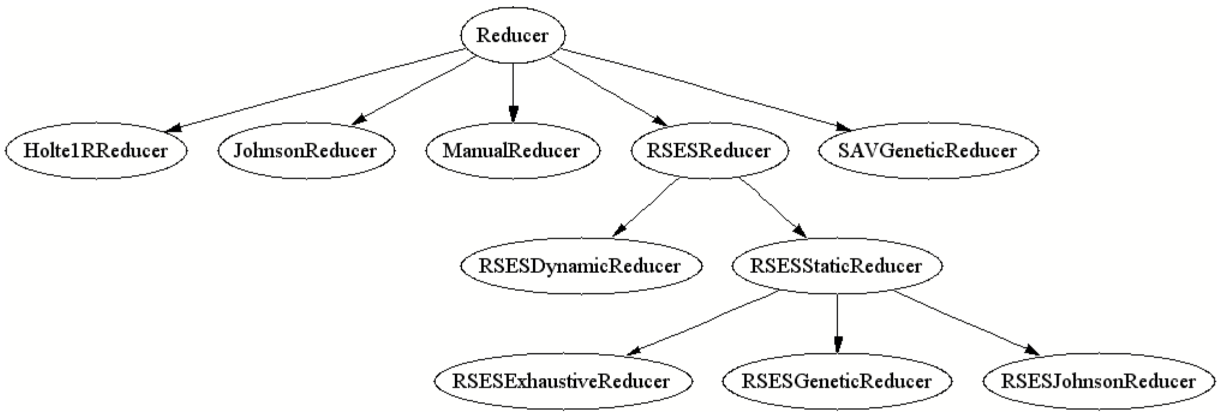

Figure 15: *Reducer* hierarchy. Subhierarchy of Figure 7.

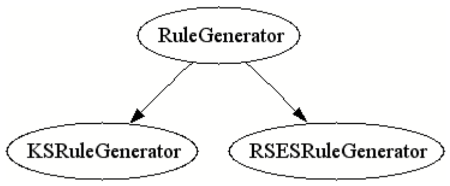

Figure 16: *RuleGenerator* hierarchy. Subhierarchy of Figure 7.

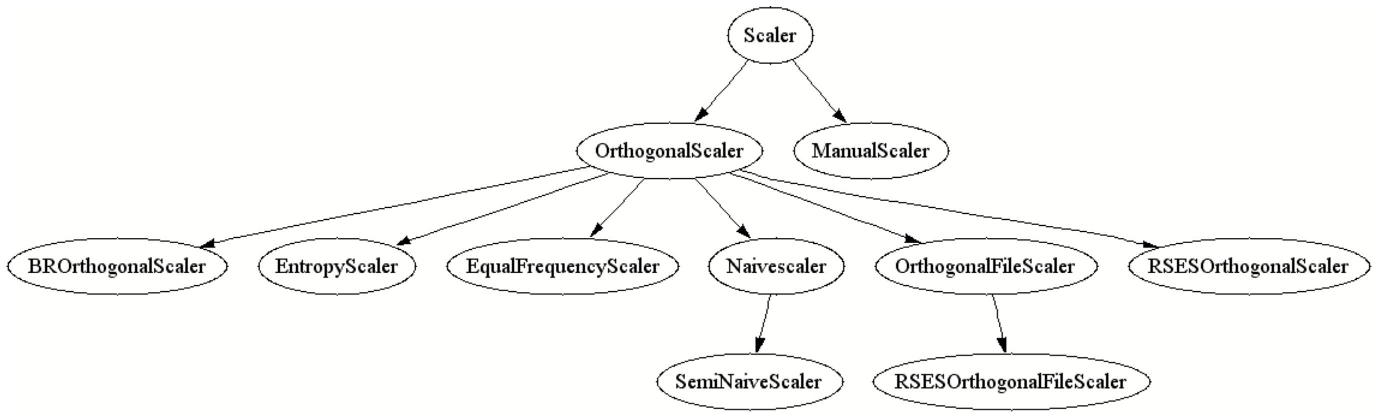

Figure 17: *Scaler* hierarchy. Subhierarchy of Figure 7.

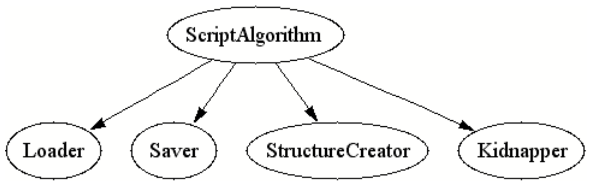

Figure 18: *ScriptAlgorithm* hierarchy. Subhierarchy of Figure 7.

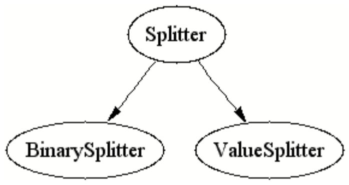

Figure 19: *Splitter* hierarchy. Subhierarchy of Figure 7.

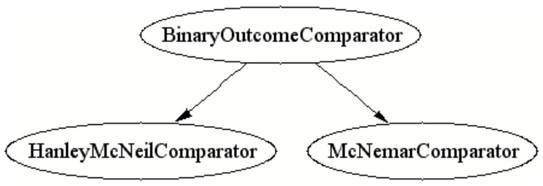

Figure 20: Miscellaneous classes.

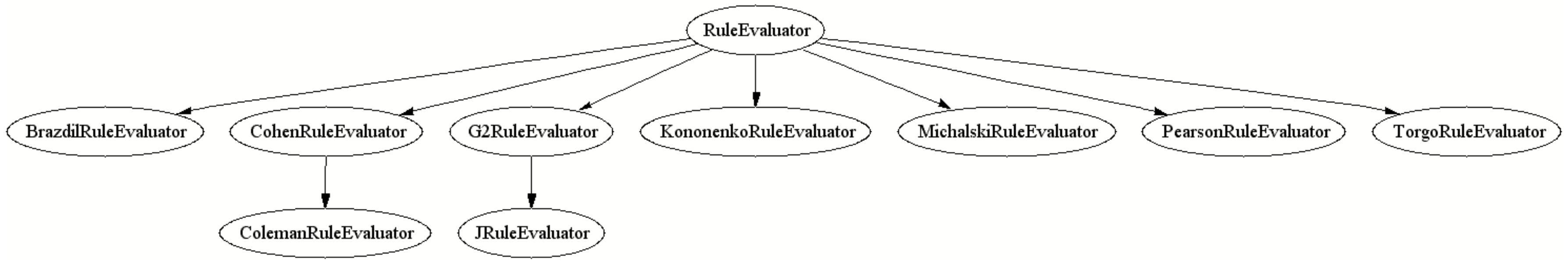

Figure 21: *RuleEvaluator* hierarchy. Subhierarchy of Figure 1.

References

[1] H. R. Arkes, N. W. Dawson, T. Speroff, F. E. Harrel, Jr., C. Alzola, R. Phillips, N. Desbiens, R. K. Oye, W. Knaus, and A. F. Connors, Jr. The covariance decomposition of the probability score and its use in evaluating prognostic estimates. *Medical Decision Making*, 15:120–131, 1995.

[2] J. G. Bazan, A. Skowron, and P. Synak. Dynamic reducts as a tool for extracting laws from decision tables. In *Proc. International Symposium on Methodologies for Intelligent Systems*, volume 869 of *Lecture Notes in Artificial Intelligence*, pages 346–355. Springer-Verlag, 1994.

[3] I. Bruha. Quality of decision rules: Definitions and classification schemes for multiple rules. In G. Nakhaeizadeh and C. C. Taylor, editors, *Machine Learning and Statistics: The Interface*, chapter 5, pages 107–131. John Wiley & Sons, 1997.

[4] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In A. Prieditis and S. Russell, editors, *Proc. Twelfth International Conference on Machine Learning*, pages 194–202. Morgan Kaufmann, 1995.

[5] The GraphViz homepage. [`http://www.research.att.com/sw/tools/graphviz/`]. AT&T Research.

[6] J. A. Hanley and B. J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143:29–36, Apr. 1982.

[7] J. A. Hanley and B. J. McNeil. A method of comparing the areas under receiver operating characteristic curves derived from the same cases. *Radiology*, 148:839–843, Sept. 1983.

[8] R. C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–91, Apr. 1993.

[9] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[10] The MATLAB homepage. [`http://www.mathworks.com/products/matlab/`]. The MathWorks, Inc.

[11] H. S. Nguyen and A. Skowron. Quantization of real-valued attributes. In *Proc. Second International Joint Conference on Information Sciences*, pages 34–37, Wrightsville Beach, NC, Sept. 1995.

[12] A. Øhrn. *Discernibility and Rough Sets in Medicine: Tools and Applications*. PhD thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, Dec. 1999. NTNU report 1999:133. [`http://www.idi.ntnu.no/~aleks/thesis/`].

[13] A. Øhrn. *ROSETTA Technical Reference Manual*. Knowledge Systems Group, Department of Computer and Information Science, NTNU, Trondheim, Norway, Nov. 1999.

[14] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[15] The ROSETTA C++ library homepage. [`http://www.idi.ntnu.no/~aleks/thesis/source/`]. Norwegian University of Science and Technology, Department of Computer and Information Science.

[16] The STLport homepage. [`http://www.stlport.org/`].

[17] S. Vinterbo and A. Øhrn. Approximate minimal hitting sets and rule templates. In *Predictive Models in Medicine: Some Methods for Construction and Adaptation*. Department of Computer and Information Science, Dec. 1999. NTNU report 1999:130. [`http://www.idi.ntnu.no/~staalv/dev/thesis.ps.gz`].

[18] J. Wróblewski. Finding minimal reducts using genetic algorithms. In *Proc. Second International Joint Conference on Information Sciences*, pages 186–189, Sept. 1995.