Chapter 19

The Design and Implementation of a Knowledge Discovery Toolkit Based on Rough Sets – The ROSETTA System

Aleksander Øhrn[1], *Jan Komorowski*[1], *Andrzej Skowron*[2], *Piotr Synak*[3]

[1] Department of Computer and Information Science, Norwegian University of Science and Technology, 7034 Trondheim, Norway
[2] Institute of Mathematics, Warsaw University, 02-097 Warsaw, Banacha 2, Poland
[3] Polish-Japanese Inst. of Computer Techniques, 02-018 Warsaw, Koszykowa 86, Poland

1 Introduction

Rough sets and Pawlak information systems [18, 19] have recently gained rather substantial scientific interest. We believe that, especially in this field, successful research requires good co-operation between theoreticians and practitioners. This can be enhanced by providing a sophisticated user environment supporting all aspects of the iterative nature of constructing models that describe or classify measurements. In response to these needs we have researched the process of using rough sets for knowledge discovery from databases (KDD). The research has had a twofold purpose: (i) to establish process patterns typical to KDD experiments, with a special emphasis on the rough set based approach, and (ii) to design and implement a programming environment with a state-of-the-art graphical user-interface (GUI) that supports construction of empirical models.

The starting point for this work was some C++ code for performing rough set related calculations that was already available in the form of the computational kernel of the RSES system developed at Warsaw University. A complete system kernel and a Windows 95/NT front-end to this has been subsequently designed and implemented at the Norwegian University of Science and Technology in Trondheim. The result is the ROSETTA system that comprises the computational kernel and the front-end. The kernel is a general C++ class library for knowledge discovery within the rough set methodology, and offers an advantageous code base for researchers to quickly assemble and try out new algorithms and ideas. ROSETTA offers a comprehensive set of software components, an environment for synthesis of propositional rules from empirical data, and constitutes a powerful vehicle for practical rough set related research and applications.

The structure of this paper is as follows. The next section presents general considerations on the nature of rough set model construction from measurements. In Sect. 3, the overall KDD process is outlined along with some of the practical issues that it raises. Based on this outline, Sect. 4 presents the requirements for ROSETTA's GUI, followed in Sect. 5 by a detailed description of the

interface and illustrated with a small but complete example in Sect. 6. Current features of ROSETTA are listed in Sect. 7. Having presented the GUI and its design, Sect. 8 summarizes the system requirements while Sect. 9 shows, from a software engineering point of view, how the design requirements have been met. Section 10 discusses data sources, formats and means of interfacing ROSETTA with databases. Finally, Sect. 11 lists some possible future extensions of the system.

Earlier summaries of the ROSETTA work in progress can be found in [15, 16]. Current information about ROSETTA can be be found at [20].

We assume that the reader is acquainted with the basic concepts from machine learning and rough set theory such as, for example, information systems, reducts, discretization and rule validation.

2 The Art of Model Construction

The purpose of rough set KDD is to find models that describe or classify measurement data. This task falls into an extensive category of pattern recognition, which, broadly speaking, can be said to be the science of constructing models that describe or classify measurements. Such models may take on a wide variety of forms according to the model construction scheme used. The choice of modelling scheme bears both to the nature of the pattern recognition problem, as well as to the purpose of the modelling task. The purpose of developing such models may be twofold. In some instances, the goal may be to gain insight into the problem at hand by analyzing the constructed model, i.e. the structure of the model is itself of interest. In other applications, the transparency and explainability features of the model is of secondary importance, and the main objective is to construct a classifier of some sort that classifies arbitrary objects well. When constructing models on the basis of labeled training data, a knowledge-based approach to such empirical modelling is to inductively infer a set of general rules that produce the desired class. This is an instance of an activity that in the knowledge systems field is collectively known as knowledge discovery or, sometimes, data mining.

Fields concerning themselves with empirical modelling necessarily have a high experimental content, both because the sought after relationships are unknown in advance and because real-world data is often noisy and imperfect. The overall modelling process thus typically consists of a sequence of several sub-steps that all require various degrees of tuning and fine-adjustments. Moreover, it may not beforehand be obvious which steps in the modelling pipeline that are required, nor which of several alternative algorithms that should be chosen for each step. As a result, the process of constructing a model is an iterated waterfall cycle with possible backtracking on the individual sub-steps. It is therefore important to have a set of tools available that render possible this type of flexible experimentation. However, a complete model construction and experimentation tool must comprise more than a collection of clever algorithms in order to be fully useful. It is needed to set the tools in an environment such that intermediate

results can be viewed and analyzed, and decisions for further processing made. Basically, an environment to interactively manage and process data is required.

3 The KDD Process

This section outlines the steps that constitute the overall process of knowledge discovery from databases. Issues springing from these have had direct bearing as overall design requirements for ROSETTA, both for the C++ class library (the kernel) and for the GUI front-end.

The overall KDD process may be broken up into several steps and phases that are iterated in a waterfall-like cycle [6], cf. Fig. 1. From a data source containing raw data, all or portions of this is selected for further processing. The selected raw data is then typically pre-processed and transformed in some way, before being passed on to the data mining algorithm itself. The output patterns from the computational mining procedure are then post-processed, interpreted and evaluated, hopefully revealing new knowledge previously buried in the data. Along the way, backtracking on each of the steps will in practice inevitably occur.

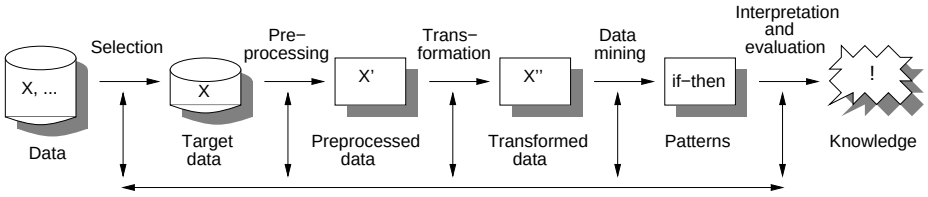


Fig. 1. Outline of the overall KDD process.

In practice, the data analyst setting up the KDD pipeline is faced with a multitude of choices that have to be made, some of which are listed below. It is in the wake of these that some central system requirements of the ROSETTA system have surfaced:

- *Which of the steps in the pipeline are necessary for the application at hand?* The topology of the pipeline may differ with regards to both the purpose of the KDD task as well as the nature and state of the data material. Moreover, each of the phases in Fig. 1 may themselves be composed of several sub-steps. A KDD toolkit should therefore offer a flexible means of constructing different pipeline topologies, preferably in a dynamic fashion so that the pipeline can have the ability to adapt to intermediate results during the course of processing, if such is requisite.

- *Which algorithms should be employed at the various stages?* There are almost always more than one way of accomplishing a certain objective. Several algorithms may be candidates for the same task, each with different trade-offs with respect to computational complexity, accuracy and spatial requirements. In essence, interchangeability is paramount so that each functional building block in the pipeline can be easily substituted with other blocks that perform the same task.
- *Which parameter set should be used for a given algorithm?* Selecting the appropriate algorithms for each step in the KDD process is usually only part of the problem. Most often, the algorithms require a set of parameters that have to be manually specified and that may have considerable impact on the performance of the algorithm. Furthermore, in many situations it is desirable to repeat the same computational process with different sets of parameters. An easy way of passing parameters that is susceptible to automation is therefore needed.
- *Which hypotheses should be pursued in the mining process?* The KDD process is both interactive and iterative of nature, with a definite exploratory flavor. That is, even though the mining steps are typically as previously outlined at large, the person performing the data analysis makes several decisions along the way as intermediate results unfold, effectively deciding the specifics of the next few analysis steps. In many cases, at each level of analysis, several competing hypotheses may be candidates for further exploration, and several algorithms yielding different results may be equally well suited. To cope with such a high degree of fan-out, it is hence imperative to be able to operate in an environment which allows one to interactively manage and process data, without losing data-navigational abilities.

3.1 The KDD Process Using Rough Sets

In a rough set framework, the natural result of an algorithm if applied to a structure is often yet another structure. For instance, a set of reducts is created when applying a reduct calculation algorithm to a decision table, and a set of rules are produced when applying a rule generation algorithm to a set of reducts. In other cases, the algorithm directly modifies the structure it is applied to.

As an illustrative example, consider the following example of a complete albeit highly simplified KDD task: A decision table residing on an external storage medium is read and pre-processed by an attribute discretization algorithm. The decision table's reducts are subsequently calculated, and the classification rules generated and exported to Prolog format. In the process, several intermediate structures are generated. The processing pipeline is displayed graphically in Fig. 2. In a slightly extended example, the pipeline would be equipped with several intermediate post-processing algorithms for weeding out "weak" reducts and rules, e.g. according to their support basis or cost. How this example pipeline can be implemented using the ROSETTA C++ class library will be shown in Sect. 9.2.

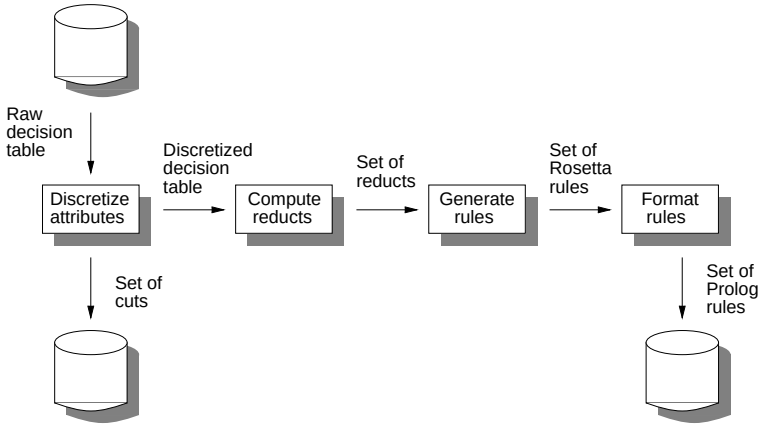


Fig. 2. Example algorithm pipeline.

4 Front-end Requirements

The above observations on the nature of the KDD process as well as general requirements on modern GUIs have lead to the formulation of the front-end requirements for ROSETTA. The resulting front-end is a GUI for interactive manipulation, construction and application of objects provided by the kernel. Windows 95/NT was chosen as the primary target platform. The most important front-end design requirements have been:

- *User-friendliness:* The front-end should be intuitive, perspicuous and easy to use, and conform to the standard Windows GUI norms. Furthermore, the GUI should be object-oriented in the sense that the objects being manipulated should be represented as distinct and individual items in the interface, with the operations that are natural to perform on the objects available directly from the objects themselves.
- *Support the overall KDD process:* In order to function as an integrated system, the front-end should cater for all steps in the modelling process; from target data selection and pre-processing, via the actual mining procedures, to post-processing, evaluation and presentation of the results.
- *Data-navigation:* During the whole modelling process, a vast multitude of different objects is likely to be generated. In order to retain navigational abilities and avoid drowning in data, it is therefore of vital importance how data is organized in the front-end. A front-end design goal has hence been to set the data in an environment where it is immediately clear which objects that exist and how they relate to each other.
- *Reflection of kernel contents:* The front-end should reflect the contents of the kernel. However, additional functionality will be added to the kernel incre-

mentally, ultimately leading to unpleasant front-end maintainability problems to deal with if not properly handled from the beginning. A front-end design goal has therefore been that incremental additions made to the kernel should reflect themselves automatically in the front-end, with little or no additional front-end programming involved.

The GUI front-end was developed using Microsoft Foundation Classes (MFC), and adheres to the MFC document/view architecture as a Multiple Document Interface (MDI) application. Present in the front-end is a commercial third-party grid control package written 100% in MFC.

5 The GUI Front-end

This section describes the ROSETTA GUI, a front-end running under Windows 95/NT that reflects the contents of the kernel C++ library. The ROSETTA GUI offers a user-friendly environment to interactively manage and process data, something that is invaluable when the model construction steps to take are not known beforehand but depend on decision-making as various intermediate structures unfold. A snapshot of a sample workspace is provided in Fig. 3.

5.1 Workspace

As previously argued, the generation of multiple intermediate structures requires that they get organized in some fashion in order for the user to retain data-navigational abilities. The ROSETTA GUI organizes its data in projects. A project is a collection of structures that all belong to or are relevant to the same modelling task. More than one project may be open at the same time. The main window in the front-end of each project is therefore a tree view. An example project tree view is seen in the upper left corner of Fig. 3, where individual objects are represented as separate icons. How the various structures relate to each other is immediately apparent from the topology of the tree. For instance, a decision table may have several other structures (directly or indirectly) derived from it, such as e.g. sets of reducts and rules, rough set approximations or other decision tables. Each branch in the tree displays a direct derivation relationship. As the modelling session unfolds, the tree is automatically updated to reflect the objects' interrelationships. The user may also manually edit the tree.

The project tree gives a bird's-eye view of the total state of the ongoing experiment. One can also zoom in and view the projects' individual member structures. Viewing of most structures is done in a matrix-like grid, in which the structures can also be manually edited or otherwise manipulated. By means of an optional dictionary, decision tables and all structural objects derived from such can be displayed to the user in terms from the modelling domain. In total, all the different views embody a comprehensive workspace.

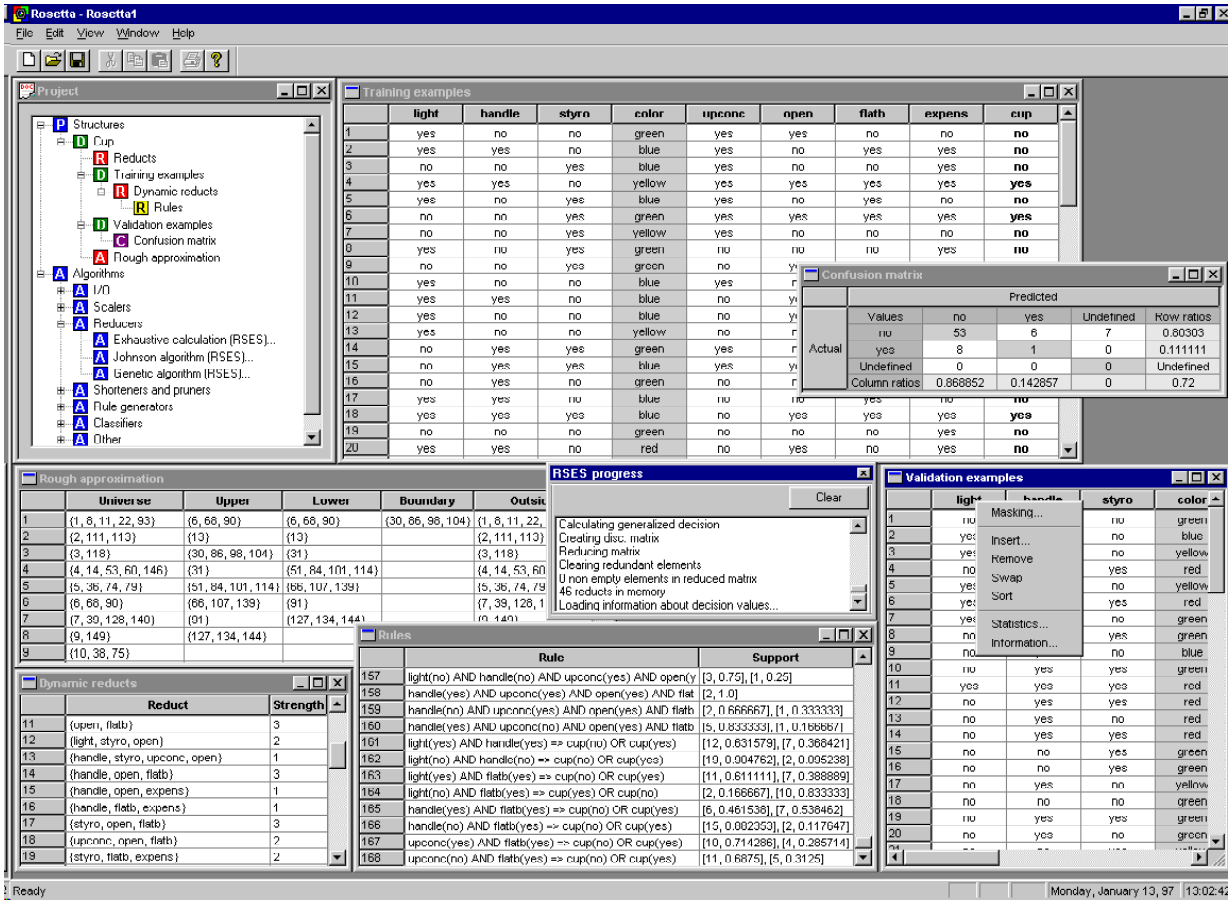


Fig. 3. Example ROSETTA workspace.

5.2 Method Invocation

A central concept in the ROSETTA GUI is that there should be maximal proximity between the object being manipulated on and the triggering of the manipulation itself. The GUI therefore heavily makes use of context-sensitive pop-up menus, invokable by right-button mouse clicks. The operations that are natural to perform on an object are hence available directly from the object itself, with the list of potential actions being dependent on the object's type and state. For instance, a menu of actions that are natural to perform on a decision table is invokable from the decision table's icon, and a menu of actions that are natural to perform on a single column (attribute) in the table is invokable from the column header in the grid view of the table.

As a supplement and alternative to the context-sensitive pop-up menus, another ROSETTA GUI feature underscoring the proximity concept is the support for drag-and-drop. Using drag-and-drop, an algorithm may be applied to a structure simply by dragging the icon of the algorithm and dropping it onto the icon of the structure to apply it to (or vice versa). The dialog box for entering parameters associated with the algorithm will then automatically pop up. For instance, to calculate the reducts from a decision table, the icon of an installed reducer algorithm may be dragged and dropped onto the icon of the decision table in the project tree. The drag-and-drop paradigm offers an intuitive and appealing way of working, and minimizes the number of user steps needed in order to perform the desired operation.

5.3 Kernel Reflection

A goal in constructing the GUI was that incremental additions made to the kernel should reflect themselves automatically in the front-end with a minimum of programming effort. By scanning a prototype pool for installed objects (to be discussed in Sect. 9), the front-end can automatically determine which structures that are available and which operations that are possible to perform on them. These can then automatically be made available to the user in the right places in the project tree and in the context-sensitive pop-up menus. Consequently, once a new structure or algorithm has been written, all that is needed to make them accessible in the front-end are the appropriate installation calls to an object manager.

5.4 Miscellaneous

The ROSETTA GUI offers an extensive set of features in addition to those already described. Among the more noteworthy of these are a prototype environment for interactive decision support on the basis of incomplete information, the ability to prematurely terminate lengthy calculations, windows displaying detailed progress messages and intermediate results, and an on-line help system.

6 Using the Front-end

To exemplify how the GUI features reflect themselves in the environment and to illustrate the working mode of the current version of ROSETTA, this section outlines how a small example KDD process can be executed from the ROSETTA GUI front-end. An example single train/test experiment is displayed in Fig. 4 and some resulting GUI objects are shown in Fig. 5 (cf. Sect. 5.1). The steps may be executed as follows:

(a) Right-click on the root in the project tree, and select *Load* from the pop-up menu. (Alternatively, use ordinary Windows document I/O). If the selected file is in an alien format, this is automatically detected and a list is presented with the installed import routines to select from. Upon loading, a decision table icon appears as a child of the root. From the table's pop-up menu, the options *View* and *Statistics* are typically selected in order to browse and get acquainted with the data. In the grid view of the table, columns/rows can be blocked out and otherwise manipulated with directly.

(b) In order to better say something about the predictive/classificatory power of the mined model, it is desirable to validate the model on a data set other than that from which it was derived. Selecting and executing an installed splitting algorithm from the origin table's pop-up menu, results in two subtables appearing as children in the project tree. One will be used to extract rules from, the other to test them on.

(c) In our example the condition attributes are all numerical and have to be discretized before proceeding. From the training table's pop-up menu, the installed discretization algorithms appear under the *Discretize* menu item. In the parameter dialog box associated with the selected algorithm, a save location for the found cuts is selected. These will be used to discretize the testing table. Upon completion, a discretized version of the training table appears as a direct descendant in the project tree.

(d) From the testing table's pop-up menu under the *Discretize* menu item, an installed algorithm capable of discretizing on the basis of the cut file produced in the previous step is selected. Upon completion, a discretized version of the testing table appears as a child in the project tree.

(e) From the training table's pop-up menu, the installed algorithms for reduct computation appear under the *Reduce* menu item. Successfully executing one of these results in a new icon appearing as a child of the training table, representing a set of reducts. (If computation of object-related reducts was selected, the reduct set icon may have a child icon itself, representing a set of decision rules. In this case, the rules have been computed as well, as this solution is computationally very cheap.) The reducts may be inspected by selecting *View* or *Statistics* from the reduct set icon's pop-up menu.

(f) It may sometimes be desirable to only proceed with some of the found reducts, especially if dealing with dynamic reducts or if we have available cost information about the attributes. Reducts may be filtered away from the reduct set by selecting an installed filtering algorithm under the *Filter* item in the reduct set icon's pop-up menu.

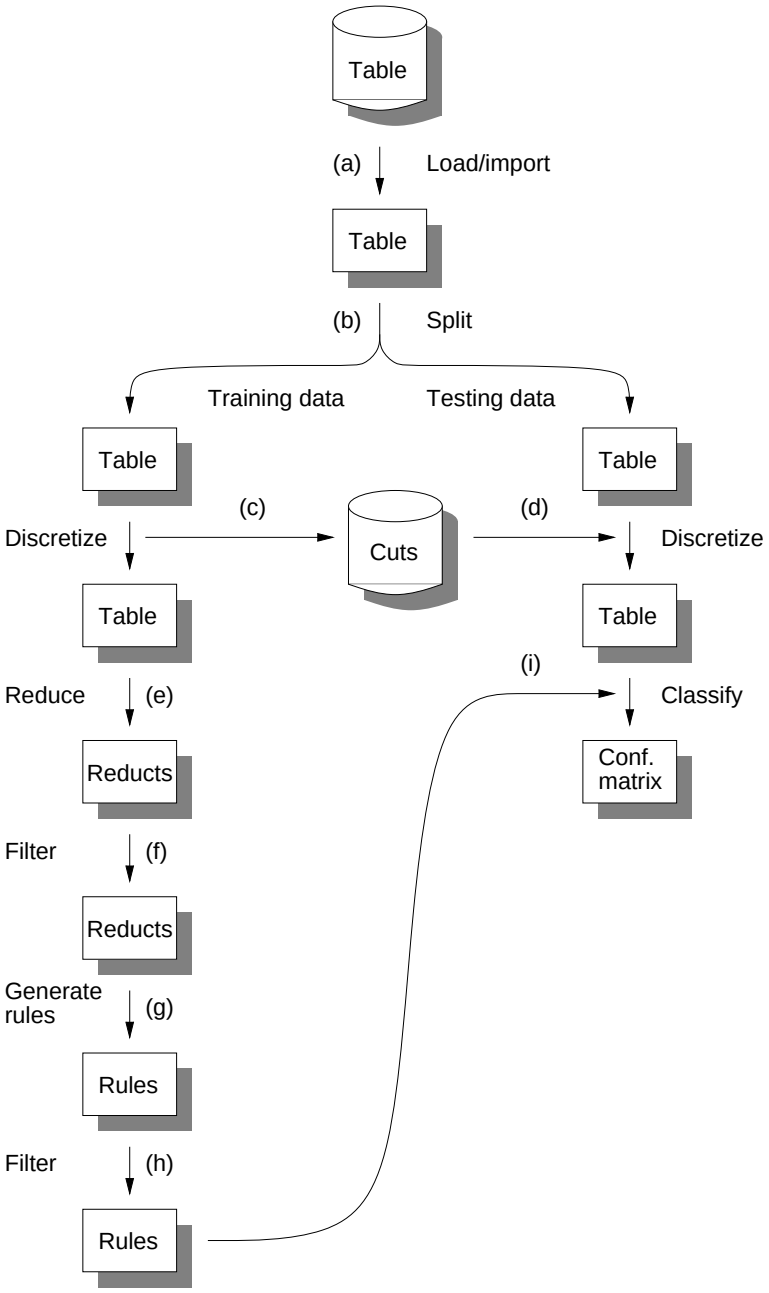
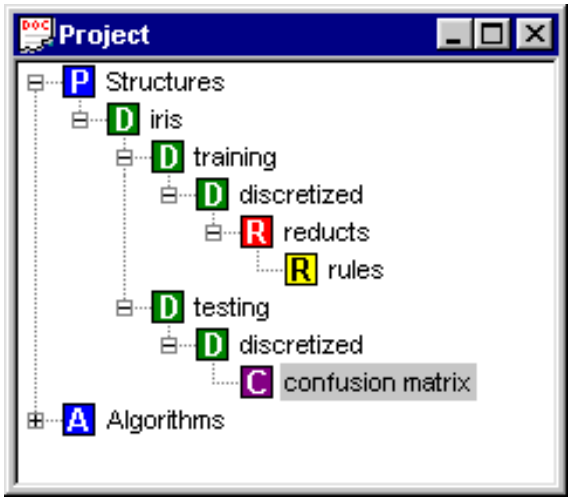


Fig. 4. An example KDD process.



rules

	Rule	Support	Probability
1	PetalLength([*, 2.6)) => Class(Iris-setosa)	22	1.0
2	PetalWidth([*, 0.8)) => Class(Iris-setosa)	22	1.0
3	PetalWidth([1.6, 1.7)) => Class(Iris-versicolor)	2	1.0
4	SepalLength([6.6, 7.0)) AND PetalWidth([1.7, *)) => Class(Iris-virginica)	4	1.0
5	SepalLength([*, 6.6)) AND PetalWidth([0.8, 1.6)) => Class(Iris-versicolor) OR Class(Iris-virginica)	20, 2	0.909091, 0.090909
6	SepalLength([6.6, 7.0)) AND PetalWidth([0.8, 1.6)) => Class(Iris-versicolor)	3	1.0
7	SepalLength([*, 6.6)) AND SepalWidth([*, 3.4)) AND PetalWidth([1.7, *)) => Class(Iris-virginica) O	15, 1	0.9375, 0.0625
8	SepalLength([7.0, *)) => Class(Iris-virginica)	5	1.0
9	SepalLength([*, 6.6)) AND SepalWidth([3.5, 3.9)) => Class(Iris-setosa)	7	1.0
10	SepalWidth([3.9, *)) => Class(Iris-setosa)	3	1.0

confusion matrix

		Predicted			
	Values	Iris-setosa	Iris-versicolo	Iris-virginica	Row ratios
Actual	Iris-setosa	28	0	0	1
	Iris-versicolo	0	22	2	0.916667
	Iris-virginica	0	1	22	0.956522
	Column ratios	1	0.956522	0.916667	0.96

Fig. 5. Some resulting GUI objects after executing Fig. 4.

(g) From the reducts we typically want to produce decision rules. This is done by selecting an installed rule generation algorithm under the *Generate rules* item in the reduct set icon's pop-up menu. Upon execution, a new icon representing the generated rules appears in the project tree as a child of the reduct set. The rules may be inspected by selecting *View* or *Statistics* from the rule set icon's pop-up menu.

(h) If we want to filter away some of the decision rules according to some criteria, this can be done by selecting an installed filtering algorithm under the *Filter* item in the rule set icon's pop-up menu.

(i) Now that we have extracted a set of classification rules from the discretized training table, we want to apply them to the discretized testing table in order to assess them. This is done by selecting *Classify* from the discretized testing table's pop-up menu. After having been prompted for which rule set to use and which installed classifier scheme to use, the objects in the testing table are classified. A new icon will appear in the project tree as a child of the discretized training table, representing a confusion matrix. This matrix reveals in which way the rules err when misclassifications occur, as well as indicate the classification accuracy and other numerical quantities of interest.

If at each step in Fig. 4 several alternative algorithms and parameter values were tried, this would simply show up as more branches in the project tree of Fig. 5.

7 System Features

This section summarizes some of the functionality currently implemented in the ROSETTA system. Some of the list items below have been mentioned in passing in previous sections, others not.

As ROSETTA was designed for extensibility, the list of features given below is likely to grow – both in different types of algorithms as well as in alternative algorithms within the same algorithmic categories. Features currently offered by the computational kernel include among others:

- Completion of decision tables with missing values according to various completion strategies. For instance, missing values may either be removed altogether, substituted with mean/mode values (possibly conditioned to the decision classes), expanded with all possible values (possibly conditioned to the decision classes), or treated as a special value in its own right.
- Computation of partitions and rough set approximations, either in the standard sense or within the variable precision model [25].
- Sampling of subtables for validation purposes.
- Discretization of numerical attributes with various discretization algorithms. Since rough set methods are well-suited for applications that take on a coarse-grained view of the world, it is typically desirable to pre-process the data so that continuous-valued attributes are converted to interval-valued ones.

Determining the cut-off points for the intervals can be done in a variety of ways. Currently implemented (or in the working) are algorithms based on Boolean reasoning [14], entropy considerations [4, 5] and χ^2-statistics [8, 11].

- Computation of reducts, both in the standard sense as well as object-related and approximate ones. As a way to deal with "noisy" data and to reveal more general patterns, dynamic reducts [2] can be computed. A genetic algorithm [22] for reduct computation is offered, as well as exhaustive computation via discernibility matrices.
- Generation of propositional rules [21]. Once the reducts have been computed, the rules are easily recovered by conceptually overlaying the reducts over the originating decision table and reading off the values.
- Filtering (shortening and pruning) of sets of reducts and rules. Several filtering criteria are currently implemented, the perhaps most interesting being filtering of reducts and rules according to their support basis, total attribute costs or a user-defined performance measure.
- Exporting of rules, reducts and tables to alien formats, e.g. to Prolog. Importing of tables from external data sources via ODBC.
- Application of synthesized rules to unseen examples by means of various classification strategies, e.g. simple majority voting or voting based on distances between possible decision classes. Confusion matrices are generated, revealing how the selected classifier performs and in what way it errs if any misclassifications take place.
- Execution of command script files, enabling automation of lengthy and repetitive command sequences.
- Support for n-fold cross-validation.

The ROSETTA GUI has been designed with user-friendliness in mind. Some of the features currently offered by the GUI include:

- Full Windows GUI conformance.
- Organization of project items in a tree structure in order to retain data-navigational abilities.
- Viewing of all structures in intuitive grid environments, using terms from the modelling domain.
- Context-sensitive menus and drag-and-drop functionality.
- Masking of attributes, enabling one to work with virtual tables.
- Automatic generation of annotations, thus documenting the modelling session.
- A prototype environment for interactive classification and guidance on the basis of incomplete information, using a selected set of synthesized rules.
- On-line help.

8 Kernel Requirements

This section describes some of the requirements that have gone into creating the kernel of the ROSETTA system. How these goals have been achieved are discussed in Sect. 9.

The ROSETTA kernel is a general C++ rough set class library offering all the most common rough set related structures and operations, and is intended for use by researchers as a toolkit for aiding rapid prototyping of new algorithms, as well as for automating sequences of steps in the overall KDD process.

A chief global requirement has been to keep a sharp distinction between the kernel of the system and the front-end. The kernel provides the relevant data structures and computations to be performed on these, while the front-end manages the manipulation and triggering of such. The kernel is independent of the front-end. This separation is important so that parts of the kernel can be embedded in other applications and employed on platforms to which the front-end cannot be ported.

A functionally central part of the ROSETTA kernel is some C++ code for performing rough set related calculations that was already available in the form of the computational kernel of the RSES system, developed at Warsaw University. A lot of resources had gone into developing this and it was natural to exploit its existence. An important design parameter for the ROSETTA kernel was therefore that it should encompass the RSES legacy code. This made it possible to produce an operational version of ROSETTA within rather short time.

Several design parameters have been emphasized in the construction and design of the kernel library:

- *Maintainability:* The object-oriented features provided by the C++ language should be fully employed together with suitable advanced object-oriented design patterns [7, 12, 13]. This promotes software component reuse and significantly contributes to the maintainability, flexibility and extensibility of the library.
- *Extensibility:* The library should cater for easy addition of new data structures and algorithms, as well as incorporation of legacy code. Notably, the system should engulf most of the core code of the RSES system, but without having to be dependent on its presence nor letting its state of design influence the overall design of ROSETTA.
- *Flexibility:* The design should allow for dynamically defined constructs and offer a versatile means of combining algorithms. Using existing library contents to try out new ideas should be easy and straightforward.
- *Modularity:* The kernel library should be composed of several individual sublibraries with strict rules concerning their interdependencies. This minimizes recompilation and clarifies the overall library architecture.
- *Usability:* The kernel's operational behavior should be consistent, and class interfaces should be uniform, complete and minimal. There should be a common way of doing things, a common idea permeating the library.
- *Efficiency:* The kernel should not be inefficiently implemented. It is a common misconception that the efficiency of numerical software written in C++ must be sacrificed on the altar of object-orientation. This is not necessarily the case. The key to avoiding this is to realize what goes on behind the scenes, and to avoid writing code in critical inner loops that incur a considerable overhead, e.g. implicit generation of temporary objects.

– *Portability:* The kernel should have no dependencies upon the front-end, and cutting-edge language constructs should, if used, be abstracted away by means of e.g. macros in order to make the kernel as compiler-invariant as possible.

Fundamental data structures such as vectors and associative maps in use by the kernel are culled from the Standard Template Library (STL).

9 The Kernel

This section describes the design of the ROSETTA C++ rough set class library and how the kernel design goals were achieved, and discusses the rationale behind some of the software engineering design decisions taken along the way. In addition, some examples of use are provided.

The ROSETTA kernel has been constructed with the design patterns and C++ language issues in [7, 12, 13] as a guide. The techniques used comprise several of those described, both behavioral, creational and structural. Some software engineering issues concerning these are really only hinted at in this section, and a more detailed treatise of the strengths and weaknesses of these patterns can be found in [7].

9.1 Main Object Dichotomy

It is worth noting that many algorithms are often implemented in terms of other algorithms. In essence, we have the notion of composite or compound algorithms. Consider for instance the pseudo-code for calculating dynamic reducts [2] given in Fig. 6. The basic formulation of the algorithm does not specify which sampling scheme to use, nor with which algorithm the subtable reduct calculation should be performed with. Coding various choices as sequences of conditional tests ultimately leads to unmaintainable code. Hence, the sub-algorithms themselves should be parameterized.

As a second motivating example, consider the case of the overall KDD process. As previously noted, the whole process is really a sequence of individually tunable steps that can be viewed as a pipeline of algorithms, with the output of one algorithm being the input to the next. In order to achieve partial automation of this process, it would be desirable to be able to chain algorithms together dynamically in a flexible fashion. Again, this leads to the design pattern of representing complex operations on objects as separate objects in their own right.

The main dichotomy in the ROSETTA kernel is between structural and algorithmic objects. Structures are simple data containers and have only methods for simple administration and access, while complex computational procedures are represented as separate algorithm objects. An example of a structural object might be a decision table, while an example of an algorithmic object would be a reducer, i.e. an algorithm that calculates a set of reducts. This separation, a variation of the Visitor design pattern [7], achieves among other:

Input: A decision table $\mathbb{A} = (U, A \cup \{d\})$ and sampling parameters θ.
Output: A set of dynamic reducts $DR(\mathbb{A})$.

$DR(\mathbb{A}) \leftarrow \emptyset$;
while $(f(\theta))$ {
    Sample a subtable $\mathbb{B} = (U' \subseteq U, A \cup \{d\})$ from $\mathbb{A}$ of size $g(\theta)$.
    Compute the set of reducts $RED(\mathbb{B})$ of $\mathbb{B}$.
    $DR(\mathbb{A}) \leftarrow DR(\mathbb{A}) \cup RED(\mathbb{B})$;
}

Fig. 6. Dynamic reduct computation pseudo-code.

- Enabling replacement or addition of new algorithms without having to modify existing structures, and vice versa. The structural classes rarely change, while defining new operations over the structures is a lot more common. When introducing a new algorithm to the system, the separation thus not only eliminates introducing bugs to the structural object the algorithm operates on, but also has a positive effect on library interdependencies and hence recompilation.
- Greater flexibility by enabling algorithms to be passed as parameters and dynamically chained together to form computational sequences.
- Simpler, smaller and more lucid class interfaces. The interfaces of the structural classes avoid getting polluted with an abundance of distinct and unrelated methods.
- Enables library clients to include into their projects only those parts of the library needed for their task at hand.

Note that the RSES legacy code does not necessarily have to employ the algorithm/structure architecture in order to be incorporated as previously described, although this would make the process more immediate.

9.2 Algorithm Application

In a rough set framework, the natural result of an algorithm if applied to a structure is often yet another structure. For instance, a set of reducts is created when applying a reduct calculation algorithm to a decision table, and a set of rules is produced when applying a rule generation algorithm to a set of reducts. In other cases, the algorithm directly modifies the structure it is applied to. In the ROSETTA kernel setting, algorithms are applied to structures by passing them through a single virtual `Apply` method on the structure object. For composite structures, the application method may be overloaded to allow for individual processing of the substructures.

The return value semantics of the application method are as follows:

```
Handle<Algorithm> algorithm[5];

// Set up the pipeline topology.
algorithm[0] = ObjectManager::GetAlgorithm(DECISIONTABLEIMPORTER);
algorithm[1] = ObjectManager::GetAlgorithm(RSESORTHOGONALSCALER);
algorithm[2] = ObjectManager::GetAlgorithm(RSESGENETICREDUCER);
algorithm[3] = ObjectManager::GetAlgorithm(RSESRULEGENERATOR);
algorithm[4] = ObjectManager::GetAlgorithm(PROLOGRULEEXPORTER);

// Set parameters.
algorithm[0]->SetParameters("filename = c:/table.txt");
algorithm[1]->SetParameters("mode = save; filename = c:/cuts.txt;");
algorithm[2]->SetParameters("seed = 1234; discernibility = normal;");
algorithm[3]->SetParameters("");
algorithm[4]->SetParameters("filename = c:/rules.pl");

Handle<Structure> structure = Creator::DecisionTable();

// Processing.
for (int i = 0; i < 5; i++)
  structure = structure->Apply(*algorithm[i]);
```

Fig. 7. Implementation of pipeline in Fig. 2.

1. If a new structural object is the natural endpoint of the algorithm, the new structure is returned.
2. If the algorithm produces no new structural object but operates on and possibly modifies the input structure itself, the input structure is returned.
3. If an error occurs, `NULL` is returned. Optionally, if exception handling is desired used, an exception is raised.

Each algorithm object keeps its own local set of parameters. Parameters are passed to algorithm objects through a standardized virtual interface. The parameter string is parsed and keyword/value pairs automatically delegated to the right accessor methods. Library clients thus have to deal with a single method only, although on the cost of several keyword/value pairs. However, this approach is more susceptible to automation through script files, and smoothes out minor syntactical issues such as case sensitivity. The accessor methods can also be called directly.

The small sample ROSETTA library C++ program in Fig. 7 completely implements the pipeline in Fig. 2, using the algorithms described in [14, 21, 22]. Note that the implicit generation of intermediate structures does not cause memory leaks in the processing loop due to the garbage collecting handle mechanism.

9.3 Handles

Perhaps the most common source of C++ programming errors relates to the management of pointers. It is often the case that some objects are shared across an object ensemble, something that opens up a wealth of potential problems concerning ownership and administration. To alleviate this, the ROSETTA C++ class library uses "smart" pointers.

A smart pointer (henceforth denoted a handle) is a templatized proxy around a traditional C-style pointer, with additional logic that ensures pointer validity and automatically takes care of memory management issues. Handles have operators overloaded to ensure that they have the same semantics as traditional pointers, and they automatically initialize to `NULL`. Moreover, a handle automatically deletes the object it refers to if it, when it leaves its scope or is reassigned, is the last handle referencing the object.

Although extremely simple, this garbage collection mechanism automatically and seamlessly minimizes or eliminates the problem of memory leaks, dangling pointers and multiple deletes, and significantly simplifies memory management and increases the safety of memory operations.

The described handle implementation only works as intended if the structures in memory form a directed acyclic graph (DAG). Pointers that introduce cycles must be ordinary "dumb" pointers. However, this shortcoming does currently not pose a problem, since a typical project defines a tree (cf. Sect. 9.7 and Sect. 5.1), a cycle-free structure. A more elaborate implementation may be done in the future.

9.4 Transitive Self-identification

The use of run-time type identification (RTTI) in C++ is a controversial issue, since it to many degrees defeats the inherent polymorphic features of the language and opens up the door to code abuse and the writing of unmaintainable code. RTTI should therefore be used judiciously, and virtual functions employed instead if possible. Yet, there are some situations where RTTI is convenient. To this end, most commercial C++ code libraries therefore supply some sorts of RTTI mechanism, and RTTI is even an integral part of the new ANSI/ISO C++ language standard.

Since many existing compilers do not yet support this language feature, the ROSETTA library offers RTTI emulation. Since public inheritance models the is-a relation, the RTTI language feature is easily emulated via the use of virtual functions. Adding the property of transitive self-identification to a class in the ROSETTA library is effortlessly done through the calls to macros. In addition, various supplementary information such as e.g. textual class descriptions may stored along with the registering of each type identifier.

9.5 Legacy Code

Incorporation of the RSES legacy code into the ROSETTA kernel has been done by means of a layering approach. By writing adapter classes (wrappers) that

inherit from abstract classes, and embedding the appropriate objects from the RSES library in these, the legacy code is effectively hidden yet made available. The derived wrapper class then performs a conversion between the interface of the abstract object and the interface of the embedded object from the RSES library.

The wrapper classes are never referenced directly as this would render the system dependent on the presence of the wrapper classes and thus indirectly upon the incorporation of the RSES code. Instead the system always operates on the level of the abstract base classes, deferring the decision of which kind of leaf-node object in the class hierarchy that actually does the work to the overloading constructs of the C++ language. This makes the system independent of the presence of RSES, and enables later substitution of the wrapper classes with other alternative implementations. Alternative implementations may also co-exist in parallel with the RSES code. Making leaf-node objects available for use throughout the system is done by installing small, empty prototype objects to a static object managing a prototype pool.

A short comment on efficiency might be in order here. Operating on the level of abstract base classes means that we are dealing with virtual functions. Since virtual functions cannot be inlined, this means that we have to incur the overhead associated with a function call e.g. for each decision table operation, most notably for every table entry lookup. This performance penalty can however usually be ignored, except perhaps for in extremely critical inner loops.

9.6 Creational Patterns

Since we wish to operate on the level of abstract objects and abstract objects cannot be instantiated, we need a creational pattern to overcome this in situations where objects need to be created. To this end, all structures have a duplication method that returns a clone of themselves. By passing a type identifier to the object manager, the pool of installed prototypes is scanned by means of the transitive self-identification procedure described earlier. The installed prototype that matches the desired object type the most closely is then ultimately duplicated and returned, i.e. inexact matches are allowed.

Apart from installation of the prototype objects, the `new` and `delete` C++ keywords are thus never used directly to create or destroy objects in the ROSETTA library. Objects are instead dynamically allocated by the creational pattern outlined above, while the previously described handle mechanism automatically takes care of the object destruction.

9.7 Navigational Issues

A modelling session may result in abundance of structures as the result of varying both alternative algorithms and their parameters. For data-navigational purposes, it is therefore important that the structures (initial, intermediate and final) can be organized in such a way that their interrelations-ships are apparent. In the ROSETTA kernel library, this is done through a recursively composed project structure.

A project structure is an object that is indirectly derived from the abstract top-level structural class, and itself keeps a list of pointers to objects of its superclass type. This organization effectively defines a tree, and allows a set of structures of different kinds to be recursively organized, with the parent/child relationship in the tree signifying which structures that have been created on the basis of which other structures. For instance, a decision table might have several reduct sets as children, and each reduct set may in turn have several rule sets as children.

Navigating in the project tree is easily done as structural objects are equipped with child pointers and offer suitable navigation methods. As an example, by passing a type identifier as a parameter to the navigation methods, this enables one to get the most immediate parent or child of a specified type (abstractly specified or not) without having to know how many intermediate structures there are between the two.

In many cases it is desirable not only to know from which structure a structure has been created, but also in what way the structure was created. To this end, structures can be annotated in the way that they can automatically be "touched" with a description of which operation that was performed on them, which algorithm that was used and with which parameter set and so on. Done systematically, this offers a means of automatically generating documentation of the modelling session. The annotation is also reversible in the sense that an extract of the annotation can be passed as a parameter set to an algorithm, and hence be used as a means of automatically regenerating the computation.

For reasons of storage efficiency, some of the navigational features are not applicable to the more simple and common library classes. For instance, adding annotation features to a single rule object would not only in most cases not only prove to be uninteresting but also spatially infeasible, as there may exist hundreds of thousands of such objects. Instead, the set of rules to which a rule belongs may be of more interest to annotate.

10 Data Sources and Formats

Prior to analysis, the data has to be read into ROSETTA in some way. For the sake of portability, the primary I/O medium of ROSETTA is plain ASCII files, assumed exported from a database of some sorts. However, ROSETTA can also interface directly with the originating database(s) by means of the Open Database Connectivity (ODBC) interface. ODBC is an open, vendor-neutral interface for database connectivity that provides access to a wide variety of computer systems, including Windows-based systems. The ODBC interface enables ROSETTA to not have to target a specific database management system (DBMS). Instead, users can add modules called database drivers that link ROSETTA to their choice of DBMS(s). Effectively, this enables ROSETTA to import tables directly from a wide variety of sources, such as e.g. spreadsheets or relational databases.

Most structural objects are exportable to non-ROSETTA formats, e.g. to Prolog. This opens up a connection to other more advanced inference engines, where also any available domain theories can be utilized.

11 Future Work

There are several ROSETTA development tasks that may possibly be executed in the future. Some of these are:

- *Scripting language:* In order to make the system fully user-extensible, the ROSETTA system may be envisioned augmented with an interpreted language for writing simulation scripts in order to write customized code, without having to physically compile code from the kernel. Such a scripting language may be fully visual of nature (as hinted at by Fig. 2), a more traditional general purpose programming language, or a hybrid of the two.
- *Data visualization:* There are typically more than one way of viewing a structure, each one emphasizing a particular aspect of the data in question. For instance, a set of rules may be visualized as a decision tree (or forest) or as a multi-dimensional scatter plot, in addition to being presented in textual form. More advanced data visualization is therefore a feature candidate for future enhancement of the ROSETTA GUI.
- *System modularization:* Presently, the kernel and the front-end of ROSETTA both reside in a single common executable. The feature of the GUI automatically scanning the kernel for installed objects is therefore not utilized to its full potential, since the executable has to be recompiled for each addition to the kernel. A future system enhancement may therefore be to split the kernel from the front-end as separate dynamic link libraries (DLLs), hence rendering possible kernel elements to be made available in the front-end in a less static fashion.
- *Cross-compilation:* Cross-compilers for porting MFC applications to other platforms exist, and this may possibly be done with the ROSETTA GUI in the future.

12 Summary

The KDD process using rough sets has been presented and analyzed. Following the requirement specifications of a sophisticated user-environment for empirical model construction, the design and implementation of a software toolkit has been outlined. The resulting toolkit covers the whole range of KDD tasks within the realm of rough sets. It consists of a general C++ class library primarily aimed at researchers for rapid prototyping, and a GUI front-end developed for knowledge discovery in an interactive setting. Issues springing from the overall KDD process have been the principal guiding design parameters for both.

The kernel class library provides a set of fundamental building blocks and the means to combine these in a flexible fashion, both for the development and testing of new algorithms and for partial automation of the overall KDD process. Various design choices made during construction of the class library have been outlined, and examples of its use been given. The GUI offers an environment wherein the fundamental tools furnished by the kernel are set. This enables interactive manipulation and creation of objects related to the KDD process.

Jointly, the kernel and the front-end offer a means to effectively and easily conduct KDD and data mining experiments within the framework of rough set theory.

In a few aspects, the work on ROSETTA is related to our previous research on providing tools and programming environments for process-oriented synthesis of logic programs [1, 9, 10].

Availability

A restricted version of ROSETTA is made publicly available on the Internet [20] for non-commercial use. The downloadable program is limited in the sense that a few algorithms are not applicable to decision tables larger than some predetermined size (currently 500 objects and 20 attributes). Details concerning conditions for obtaining a full-fledged version of ROSETTA are to be announced. By Fall 1997, (restricted) ROSETTA was downloaded by nearly 300 people. To our knowledge, its use is reported in [3, 23, 24, 17].

Acknowledgments

The authors of this article would like to thank all members of the teams at Warsaw University and at the Norwegian University of Science and Technology in Trondheim involved in the design and implementation of the ROSETTA system, in particular Jan Bazan, Adam Cykier, Hoa S. Nguyen, Son H. Nguyen, Jakub Wroblewski, Knut Magne Risvik, Daniel Remmem, Jørn E. Nygjerd, Ivan Uthus and Merete Hvalshagen. They have all contributed to what the ROSETTA system is today.

This research is supported in part by the European Union 4th Framework Telematics project CARDIASSIST, by the Human Capital and Mobility Norwegian Research Council contract #101341/410, by the Norwegian Research Council grant #74467/410, and by the Norwegian Research Council grant for Cooperation with Central Europe. This work is also partially supported by the National Committee for Scientific Research in Poland under grant #8T11C01011 by the ESPRIT project 20288 CRIT-2.

Special thanks are due to the Norwegian Research Council for the sabbatical fellowship and to the Department of Automatic Control at Lund Institute of Technology that kindly provided its facilities and a stimulating environment during Jan Komorowski's sabbatical.

References

1. Ballandby, P.: Towards automating software design in the partial deduction framework. Master Thesis, Norwegian University of Science and Technology (1992)
2. Bazan, J., Skowron, A., Synak, P.: Dynamic reducts as a tool for extracting laws from decision tables. In: Z. W. Ras, M. Zemankova (eds.), Proceedings of the

Eighth Symposium on Methodologies for Intelligent Systems, Charlotte, NC, October 16-19, Lecture Notes in Artificial Intelligence **869**, Springer-Verlag (1994) 346–355
3. Ciupke, K.: Diagnosis of unbalances in rotating machinery (manuscript)
4. Dougherty, J., Kohavi, R., Sahami, M.: Supervised and unsupervised discretization of continuous features. In: Proc. of the Twelfth International Conference on Machine Learning, Tahoe City, CA, USA, Morgan Kaufmann (1995) 194–202
5. Fayyad, U., Irani, K. B.: Multi–interval discretization of continuous attributes as preprocessing for classification learning. In: Proc. Thirteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann (1995) 1022–1027
6. Fayyad, U., Piatetsky-Shapiro, G., Smyth, P.: The KDD process for extracting useful knowledge from volumes of data. Comm. ACM, **39/11** (1996) 27–34
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns – elements of reusable object-oriented software, Addison-Wesley, New York (1995)
8. Kerber, R.: ChiMerge: discretization of numeric attributes. In: AAAI-92 Proc. Ninth National Conference on Artificial Intelligence, AAAI Press/MIT Press (1992) 123–128
9. Komorowski, J., Trcek, S.: Towards refinement of definite logic programs. In: Z.W. Ras, M. Zemankova (eds.), Proc. of the Eight International Symposium on Methodologies for Intelligent Systems, Charlotte, NC, October 16-19, 1994, Lecture Notes in Artificial Intelligence **869**, Springer–Verlag, Berlin (1994) 315–325
10. Komorowski, J.: Automating the design and maintenance of Prolog programs with REFINERY. Technical Report, Norwegian Institute of Technology, also accepted for poster presentation at PAP'96 – Practical Applications of Prolog, London, April 23–25th (1996)
11. Liu, H., Setiono, R.: Discretization of ordinal attributes and feature selection. In: Proc. Seventh International Conference on Tools with Artificial Intelligence, Washington DC (1995) 388–391
12. Meyers, S.: Effective C++: 50 specific ways to improve your programs and designs. Addison-Wesley (1992)
13. Meyers, S.: More effective C++: 35 specific ways to improve your programs and designs. Addison-Wesley (1996)
14. Nguyen, S. H., Skowron, A.: Quantization of real-valued attributes. In: P.P. Wang (ed.): Second Annual Joint Conference on Information Sciences (JCIS'95), Wrightsville Beach, North Carolina, 28 September - 1 October (1995) 34–37
15. Øhrn, A., Komorowski, J.: ROSETTA – a rough set toolkit for analysis of data. In: P.P. Wang (ed.): Proceedings of the Fifth International Workshop on Rough Sets and Soft Computing (RSSC'97) at Third Annual Joint Conference on Information Sciences (JCIS'97), Duke University, Durham, NC, USA, Rough Set & Computer Science **3**, March 1–5 (1997) 403–407
16. Øhrn, A., Komorowski, J., Skowron, A., Synak, P.: A software system for rough data analysis. Bulletin of the International Rough Set Society **1/2** (1997) 58–59
17. Øhrn, A., Vinterbo, S., Szymański, P., Komorowski, J.: Modelling cardiac patient set residuals using rough sets. In: Proc. AMIA. Annual Fall Symposium (formerly SCAMC), Nashville TN, Oct. 25–29 (1997) 203–207
18. Pawlak, Z.: Rough sets. International Journal of Computer and Information Sciences **11** (1982) 341–356
19. Pawlak, Z.: Rough Sets – Theoretical Aspects of Reasoning about Data. Kluwer Academic Publishers, Dordrecht (1991)
20. The ROSETTA WWW homepage, `http://www.idi.ntnu.no/~aleks/rosetta/`

21. Skowron, A.: Synthesis of Adaptive Decision Systems from Experimental Data. In: Aamodt, A. , Komorowski , J.(eds.) Proc. Fifth Scandinavian Conference on Artificial Intelligence, Trondheim, Norway, May 29–31 (1995). Frontiers in Artificial Intelligence and Applications, **28** IOS Press, Amsterdam (1995) 220–238
22. Wróblewski, J.: Finding minimal reducts using genetic algorithms (extended version). In: P.P. Wang (ed.): Proc. of the Second Annual Joint Conference on Information Sciences (JCIS'95), Wrightsville Beach, North Carolina, 28 September - 1 October (1995) 186–189
23. Zhang, Q., Han, Z., Wen, F. (1997): A new approach for fault diagnosis in power systems based on rough set theory. In: Proc. International Conference on Advances in Power System Control, Operation and Management (APSCOM-97), Hong Kong, China, November 11–14 (1997)
24. Zitner, D., Paterson, G. I., Fay, D. F.: Methods in health decision support systems: Methods for identifying pertinent and superfluous activity. In: Tan, J. , Sheps, S. (eds.), Health Decision Support Systems, Aspen Publishers (1997)
25. Ziarko, W.: Variable precision rough set model. Journal of Computer and System Sciences **46** 39–59

This article was processed using the LaTeX macro package with LMAMULT style