GrailKnights: An automaton mass manipulation package for enhanced pattern analysis.

by

Hercule du Preez

Department of Computer Science
University of Stellenbosch
Private Bag X1, 7602 Matieland, South Africa

Supervisor: Dr L. van Zijl

March 2008

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: ..... Date: .....

Abstract

This thesis describes the design and implementation of an application named Grail-Knights that allows for the mass manipulation of automata, with added visual pattern analysis features. It comprises a database-driven backend for automata storage, and a graphical user interface that allows for filtering the automata selected from the database with visual interpretation of visible patterns over the resulting automata.

Uittreksel

Hierdie tesis beskryf die ontwerp en implementering 'n rekenaarpakket genaamd GrailKnights, wat massa-manipulasie van outomate toelaat met bykomende visuele patroonontledingseienskappe. Dit bestaan uit 'n onderliggende databasis vir die stoor van outomate en 'n grafiese gebruikerskoppelvlak wat die filtrering van geselekteerde outomate uit die databasis toelaat, met visuele interpretasie van sigbare patrone oor die resulterende outomate.

Acknowledgements

I would like to express my sincere gratitude to the following people who have contributed to making this work possible:

- Dr Lynette van Zijl of the University of Stellenbosch as my supervisor and editor,
- Gwen Raitt for multiple proofreadings and constructive criticisms and
- Lesley Anne du Preez for her love and support.

Contents

List of Figures

Chapter 1

Introduction

Regular language algorithms have a wide variety of applications. These applications include computational linguistics, state-machine compilers, textual pattern matching and data compression, to name but a few.

Regular languages can be described by regular expressions or by finite automata (see Appendix A), among others. That is to say, regular expressions and finite automata can be used to write down finite regular languages in a finite way. Finite automata can be used by a computer to recognise a finite language. Various algorithms exist to manipulate both these representations and to investigate the properties of the regular languages represented by them. Over the last decade, a number of regular language packages has been developed, each of which incorporates a selection of these algorithms into a unified package. The packages differ in a number of respects:

- some packages (such as FIRE Station [13]) offer graphical user interfaces, while others are command line driven (such as Grail+ [31]);
- some packages can only operate on one finite automaton (or regular expression) at a time, while others allow for a batch mode where the same operation can be performed on many automata; and
- some packages are centered around teaching the concept of automata and their algorithms (such as Exorciser [39], Kara [32] and JFLAP [34]), while others are research orientated (such as AUTOMATE [9]).

However, in regular language research, an experimental scenario that often occurs is one where a huge number of finite automata are randomly generated and then manually analysed for certain patterns. In the current offering of automata manipulation packages, this scenario leads to an unordered collection of automata which are sequentially streamed through the package which outputs a multitude of text files. This makes pattern analysis of the results a laborious task and the user often needs to program large scripts and/or resort to manual pattern analysis.

We designed and implemented an automata mass manipulation toolkit, called GrailKnights[1], with the following main goals:

[1] The references to the Grail in GrailKnights are used because the default automaton manipulation package used by this application is known as Grail+ [31].

- it should collect finite automata and regular expressions into a coherent format for easy access and manipulation,
- it should offer a user-friendly and intuitive visual interface through which any number of filters can be applied to select automata in a database and results displayed in a format which allows for easier visual pattern analysis,
- the amount of programming required by the user should be kept to an absolute minimum, and
- the toolkit should be generic, in the sense that it should be possible to combine it with other third-party automata manipulation packages.

We present an overview of other automata manipulation packages in Chapter 2. Chapter 3 discusses the design methodology used in creating the GrailKnights system. This is followed by the implementation details of the GrailKnights system in Chapter 4. Chapter 5 analyses the results of the implementation, by considering a detailed large example of the use of the GrailKnights toolkit. We conclude in Chapter 6.

Chapter 2

Literature Survey

This chapter consists of two sections: a discussion of existing automata packages and an analysis of human-computer interaction as it pertains to our proposed system.

In discussing existing automata packages, we list and discuss applications and packages that can and have been used as automata research packages. Most of the packages contain a library of automata conversion and manipulation algorithms, such as the algorithm to convert a nondeterministic finite automaton (NFA) to a deterministic finite automaton (DFA) and the algorithm to minimize a given DFA. Most packages have some type of user interface, though in some cases this is little more than a command driven interface.

The human-computer interaction section briefly examines the current literature about the interaction between a user and the computer, with focus on the visual or graphical user interface.

2.1 Existing automata and automata manipulation packages

2.1.1 AMoRE

AMoRE [28] or Automata Monoids and Regular Expressions, is a library of finite automata related computational procedures written in the C language. Its development started in 1986 and it has had a vast number of contributors since. It is one of the older packages but it does include both a textual interface and an automaton state graph display mechanism along with many conversion algorithms.

While this application can manipulate and display a single automaton, it cannot manipulate or display more than one at a time, which is not ideal for the mass manipulation of automata.

2.1.2 AUTOMATE

AUTOMATE [9] is an application that handles 'symbolic computation on finite automata, extended rational expressions and finite semigroups'. AUTOMATE originated in 1984 and has since been successfully used as a research tool. It, like AMoRE, is written in C for portability and efficiency and has been used on various early versions of UNIX, DOS and even on the Macintosh.

This package has a textual (that is, non-graphical) user interface through which regular expressions and finite automata can be manipulated. However, these manipulations can only be done on a single expression or automaton at a time.

2.1.3 FIRE Engine

FIRE Engine [42] is a toolkit for **FI**nite automata and **R**egular **E**xpressions. Its creation marks the beginning of similar FIRE packages such as FIRE Lite and FIRE Station (see below). FIRE Engine, unlike AMORE and AUTOMATE, has no interactive user interface, as its intention is to be integrated into applications programmatically. Algorithms implemented are discussed in full by Watson [43].

Although the algorithms were implemented with efficiency in mind, this package does not directly allow for the mass manipulation of automata. While it does allow the researcher to do these manipulations indirectly, the researcher is required to do a certain amount of programming to achieve the desired results.

2.1.4 FIRE Works and FIRE Station

FIRE Works and FIRE Station [13] are separate packages, both created by the same author, to form a single working unit. FIRE Works is a toolkit used for the construction and manipulation of regular languages, regular expressions and finite automata. FIRE Station is used to visualize and manipulate the graph data structures of the underlying FIRE Works.

The theory behind the FIRE Station package is that the package creates a finite automaton in such a way as to guarantee the retrieval of the original, human-readable, regular expression that created it. While it is possible to obtain a regular expression from a finite automaton, there are many such regular expressions that yield the same automaton and the resultant regular expression is generally larger than necessary. The FIRE Station package, however, does not cater for mass manipulation of the said automata. While the package can display a visual representation of an automaton, it cannot compare the separate automata.

2.1.5 Grail+

Grail+ [31] provides a pair of interfaces: one is, like FIRE Engine, a set of raw C++ class libraries that can be used to manipulate automata programmatically. The other is a set of 'filter' programs, which, as in the traditions of UNIX, change a given input into an output depending on the algorithm used. UNIX piping can be used to create a chain of filters for more complex calculations. This is the real power behind Grail+, as the filters are easy for any UNIX user to use and understand, therefore little time is wasted in learning a new interface or programming. This, coupled with the large number of algorithms made available by Grail+, has made Grail+ a popular research tool.

The mass manipulation of automata can be programmatically implemented by Grail+ but the analysis of the output has to be done manually.

2.1.6 MERLin

The MERLin [41] package (Modeling Language for Regular Languages) is built using LEDA [29], a C++ library for implementing graph algorithms. The MERLin package serves as a graphical user interface to the Grail+ libraries mentioned earlier and includes additional functionality of its own. This includes creating one or more automata (randomly, interactively or from a file) and then saving the results to files. The Grail+ function calls are then applied to these files to produce output files. These files are saved in a Grail+ format. MERLin also allows for parallel processing.

While MERLin is a step in the right direction where mass calculations on automata are concerned, it does not allow for display and therefore visual analysis of multiple automata simultaneously.

2.1.7 FSA Utilities

The FSA (**F**inite **S**tate **A**utomata) Utilities [23] is a toolbox of algorithms used to manipulate finite state automata and finite state transducers. It is written in SICStus Prolog. This set of utilities aims at a toolbox useful for computational linguistics, such as finite-state techniques being used for computational phonology and morphology, efficient dictionary lookup and part-of-speech tagging, natural language processing, techniques for parsing ill-formed input and speech recognition.

FSA Utilities can generate LaTeX and Postscript images of automata or use external general graph drawing programs (these include VCG [35] and daVinci [25]) as well as being able to manipulate a given automaton graphically using the *tk*-widget [2]. Unix piping can be used to allow for mass manipulation of automata. However, there is no means of analysing the mass output without resorting to doing it manually.

2.1.8 FSA Toolkit

The FSA (Finite State Automata) Toolkit [23], not to be confused with FSA Utilities above, is a collection of algorithms for creating and manipulating weighted finite state automata. While it is written in C++, it does support a Python interface. Among other things this toolkit aims to be a comprehensive library of algorithms with low computational costs and an easy to use interface. The toolkit, while having algorithms that may execute at fast speeds, does not include a graphical user interface nor any method for comparing multitudes of automata with one another.

2.1.9 Vaucanson

Vaucanson [27] has the capacity to deal with automata whose labels belong to various algebraic structures as well as to allow the programming of automaton manipulation algorithms such that the programming syntax is close to the mathematical representation of the algorithm. Vaucanson uses rich data structures without negatively impacting the performance of the algorithms.

Vaucanson is a C++ generic library for finite state automata and therefore does not have its own graphical output or graphical user interface but instead relies on external programs for this functionality.

Package	Properties	Mass-manip.
AMoRE	library, automata monoids, textual interface	no
AUTOMATE	library, finite semigroups, textual interface	no
FIRE Engine	library	no
FIRE Station/Works	library, loss-less FA creation, GUI	no
Grail+	library, easy use in UNIX	no
MERLin	parallism, multiple automata experiments	partly
FSA Utilities	library, finite state transducers, GUI	no
FSA Toolkit	library	no
Vaucanson	library, algorithm creation	no

Figure 2.1: Summary of Automaton Packages

2.1.10 Automaton Packages Synopsis

In summary, (see Fig. 2.1) while most existing automata packages can apply an algorithm to a single input, few allow for the mass manipulation of automata, that is, applying an algorithm or a series or algorithms to a batch of inputs. While most can display a single automaton state graph, there is a distinct lack of applications that can *visually* (or otherwise) compare the properties of large numbers of automata.

2.2 Human-computer interaction

The field of human-computer interaction (HCI) is a vast multidisciplinary science. It is the combination of social and behavioural sciences and computer and information technology. A more appropriate name for HCI would be human-machine interaction, since the field does not only concern the interaction between a human user and a computer but also includes other technological devices from printers to cellular telephones. HCI professionals attempt to underline how humans make use of technological devices and how the devices can be made more useful and usable, that is, user-friendly. HCI as a field, has become fragmented across multiple disciplines which reflects the breadth of its scientific foundations. In an effort to unify the many theories of HCI, sources are usually compilations of the multiple theories and experiments by HCI professionals in various fields [8, 20, 24, 26, 40].

This thesis focuses on design practices for the computer visual interface or what is normally termed the graphical user interface. To this end, the HCI field offers designers three main forms of guidance: guidelines, principles and theories.

2.2.1 Guidelines, principles and theories

A guideline document helps to create interfaces that are consistent in terminology, appearance and action sequences between different applications for a platform or programming language for which the guideline document is written. An example of such a document is given by Johnson [22], which lists 82 common mistakes made by software developers, with suggestions or guidelines on how to circumvent those mistakes.

While guidelines tend to be specific or focused, principles are, on the other hand, broader and more widely applicable. Johnson [22], in addition to the guidelines, lists the following as first principles for the software designer:

- *Focus on users and their tasks, not the technology.* The designer should focus on the intended users and intended users' tasks before taking technology into account. This requires that the designer first find out what the product or service is for, what activities or solutions it should provide and what problems the users have currently. The designer should also discover the relative skill and knowledge of intended users, how users conceptualize the problem and/or solution and what the users' preferred methods of solving a specific problem are. This, and more, user orientated information should be discovered before the decision is made on which programming language to use.

- *Consider function first, presentation later.* Software developers should consider the conceptual role or function (not implementation) of the user interface before considering how it should appear visually. This also includes choosing what concepts will be exposed to users and how the users will create, view and

manipulate data as well as what options will be available to the users through the software. Creating a conceptual model is important for further design work.

- *Conform to the users' view of the task.* How users conceptualize a problem is important. The designer should strive to create an intuitive interface by not imposing arbitrary restrictions on the user, such as allowing a maximum of ten characters for a first name. The designer should employ the users' vocabulary and not his/her own. The designer should conceal program internals from the user. Lastly, it is important to find a balance between power and complexity since, for example, a overly complex user interface would lead to confusion, while a simplistic one may not have the functionality required by the users.

- *Do not complicate the users' task.* The ideal user input is no input at all. Unfortunately, input is still required, but it should be held to a minimum. Decreasing user interaction but keeping the same power is difficult, but can be done through having default values, adding user customization and using multi-page dialogs known as wizards. Not giving the user extra problems to solve also decreases the complication of tasks. An example of such a complication is asking the user to input a random generator seed when such a seed could be generated by the program itself.

- *Promote learning.* In order to encourage easy learning of a system, one should remove ambiguity in text, graphics and layouts; have a consistent interface overall; think from the users' perspective and provide a low risk environment. A low risk environment is one in which changes can be easily undone and mistakes quickly found and corrected. The methods mentioned here are far from exhaustive.

- *Deliver information, not just data.* Information is data correctly presented. This presentation can be influenced negatively by incorrectly directing user attention; not creating the software for the medium on which it is shown and, finally, lack of attention to details. To ensure the proper delivery of information, the screen should remain under the control of the user.

- *Design for responsiveness.* Examples of poor responsiveness include poor or nonexistent feedback to user, time consuming operations that block other activities, providing no clue as to how long an operation will take and periodically ignoring user input to do internal housekeeping. It should be noted that responsiveness is not the same as performance (speed of execution) since a program may be responsive but still take a hour to complete a task. The converse of this is that a high performance task may take ten minutes for the same hour long task but during those ten minutes it seems to the user as if the program has crashed, that is to say, it is unresponsive.

- *Try it out on users, then fix it!* Last but not least, the software should be tested on the intended users to uncover usability problems and then the problems should be repaired in the next version of the software.

Schneiderman and Plaisant [36], similarly to Johnson, place emphasis on users: that is, what their skill levels are and on what tasks the users want performed. Schneiderman and Plaisant follow these ideas with the eight golden rules of interface design:

- *Strive for consistency.* There should be a consistency of action sequences, terminology, colour, layout, capitalization and so on. Exceptions should be comprehensible and limited in number, for example, a delete function should require a confirmation and be highlighted in a different colour to other action sequences.

- *Cater for universal usability.* The designer should make possible the alteration of content to meet the needs of different users. Changes should take into account the expertise of the user on the system (novice to expert), the users' age, any user disabilities and the users' technological aptitude.

- *Offer informative feedback.* This principle states that for every user action there should be some acknowledgement by the system. The severity of the action should be reflected in its acknowledgment by the system, that is, for minor or frequent actions responses should be minimal, while responses to important operations should be large.

- *Design dialogs*[1] *to yield closure.* While this principle may seem part of the previous principle, it is important enough to stand on its own. Concluding a series of actions with a dialog, informs the user that the series has completed. To illustrate, imagine filling in a form on a website, submitting it and finding the browser back at the website's home page. Without a concluding page, user confusion arises as to whether or not the action completed successfully or not.

- *Prevent errors.* The designer should design an interface so that the users' errors are minimised. Examples of how this may be done include greying out unusable buttons or menus and only allowing numerical values in a text field that requires a number.

- *Permit easy reversal of actions, that is, undo.* For any user action there should be an equal and opposite undo action. Such a feature relieves user anxiety and encourages exploration of unknown options. Mistakes may only be made apparent after an action or several actions later.

- *Support internal locus of control.* This principle states that the user has the ultimate control of the application. An application that does not adhere to the users' will, will lead to frustration and dissatisfaction with the application. Examples of such factors that lead to user frustration include surprising interface actions, long sequences of input data, inability to obtain information and inability to produce the desired action.

[1] A dialog is a popup frame extensively used in windowing environments (the various Windows and graphical Unix-based environments) to display an important message which the user must acknowledge before proceeding, usually by pressing the 'OK' button. More complex dialogs exist but this is the simplest form.

- *Reduce the users' short-term memory load.* The user should not need to remember specific values to successfully complete an action. Such values are for example, long lists of items or a specific number from a different page and so forth.

Some of these golden rules above are succinct mirrors of the guidelines given by Johnson [22].

Lastly, theories can be characterised as the following:

- Theories are based on the low-level principles mentioned above.
- Theories explain or predict the specifics of guidelines.
- Theories are thoroughly tested and reliable.

Theories are as numerous as the guidelines they predict and so no theories will be mentioned in this work specifically. More generally though, theories can be grouped into different categories such as

- descriptive or explanatory theories and
- predictive theories.

Another method of grouping is as follows:

- motor-task performance,
- perceptual activities, and
- cognitive aspects theories.

More information on theories can be found in, among others, [8, 20, 24, 26, 40].

In summary, the design and implementation of the GrailKnights system will be guided by the principles listed above while also taking into account the guidelines mentioned by Johnson.

2.3 Object oriented programming

Before continuing into a discussion of the design of the project, one needs take into account the underlying design principles and goals given by object orientation. Many sources [6, 7, 14, 30] describe these goals and principles. The goals are

- robustness,
- adaptability and
- reusability.

Programmers should strive to make their programs as robust as possible which implies that the program produces correct results for given inputs and also handles unexpected inputs correctly. An adaptable program is one that can run on changing hardware and operating systems with the least amount of change. This is also known as the programs evolvability or portability. A program should be written in such a way so that it can be used in various applications without much (or any) change. This is called a programs reusability. The design principles are as follows:

- abstraction,
- encapsulation, and
- modularity.

One abstracts a complex system down to its most fundamental parts and describes those parts concisely. The principle of encapsulation states that a component should not reveal its internal workings. Finally, the principle of modularity states that in a system of multiple components, the components are divided into separate functional units. In the design chapter we will use these object orientated goals and principles to design the various components of the GrailKnights project.

2.4 Visual pattern analysis

The goal of the designers of modern visual pattern analysis systems is to help users discover which questions to ask about the data in question. To this end, visual tools are created in order to: give an overview, explore rapidly, test hypotheses and share the results with peers. One approach to creating such a visual tool is called *dynamic queries* [5] which allows interactive exploration of the data (the data is generally stored in a database). The central idea is that users change various graphical widgets (for example, buttons, sliders and check boxes) in order to update a graphical display which updates quickly. This is instead of using complicated syntactical commands to accomplish similar operations.

It should be noted that a visualization should match the data being visualized. For instance, treemaps [5] have been shown to represent hierarchical data (such as directory structure and network structure) well. On the other hand, multidimensional numerical data is easily shown on a starfield display by mapping two dimensions to the X and Y coordinates and mapping other dimensions to visual attributes such as colour, orientation, intensity, shape or size (width or height). If an information visualization display holds more than a few thousand items it typically becomes overcrowded and it becomes difficult to observe trends, clusters, outliers or gaps in the data. Aggregations or groups of data points may be used in such cases to provide a summarization or overview of the data.

Visualization techniques should rely as much as possible on pre-attentive graphical features. These are features that may be recognized at a glance without effort. An example of a pre-attentive graphical feature is spotting a red dot between a myriad of blue dots, while an example of a non-pre-attentive feature is text reading.

There is a small set of visual features that may be used in a pre-attentive way [18]: line (for example underline or strike-through), orientation (angle and placement), length, width, size, curvature, number, terminators (for example symbol(s) indicating the end of a line), intersection, closure (for example a shape being surrounded by another shape), colour (hue), intensity, flicker, direction of motion, binocular luster (texture), sterioscopic length, 3D depth cues and lighting direction. It is also noted that in controlled configurations these features may be processed pre-attentively, not that they always will be processed pre-attentively. For example only five to seven well chosen colours may be processed pre-attentively. Adding more colours increases search times. In addition, the use of many pre-attentive features may interfere with one another so that in practice only two or three features may be used together.

Visual pattern analysis, in short, is the ability that humans have to quickly and effortlessly recognise patterns in a group of items, as well as to recognise items that are not part of an overlying pattern. An example of this is a scatterplot where there is a large group of items with a few outliers. The outliers are quickly recognised, while if one were to read through a large list or table of numbers, these outliers would not be so apparent. This is the power behind visual pattern analysis: while computers excel at doing a myriad of calculations accurately and near instantaneously, humans excel at pattern recognition.

Chapter 3

GrailKnights: Design

This chapter describes the design goals and design principles used in the development of the GrailKnights package. The human computer interaction guidelines and principles from the previous chapter were used during the design phase of the GrailKnights project. The main goals of the GrailKnights project are as follows:

- to provide for the mass manipulation of automata,
- to provide methods to indicate visually the relationships between automata as well as display the automata themselves,
- to require minimal programming by the user, and
- to be modular so that advanced users may add their own automata manipulation algorithms or packages, as well as databases of their choice.

The following are sub-goals of the GrailKnights package. These goals may not be design specific and some may come into play in the implementation chapter. However, they are important enough to bear in mind while designing.

- **Distributed.** A distributed system implies that some components of the package may be on different physical machines. The advantages of this are remote access as well as processing optimisation where long processes may be done on a faster machine and commands given from a slower one. The drawbacks of a distributed system are the network traffic congestion, less robustness in the case of a centralised component (if the centralised component fails the entire system may fail), and distributed systems have a higher security risk. Since security is not one of the goals of GrailKnights, we may ignore it.
- **Data transparency.** The target users of the GrailKnights package are automata researchers and as such they should have direct access to the data. While this seems to complicate the user's task, it increases the degree of control over the data as well as 'conforming to the user's view of the task'. This transparency should also extend to any external packages in that any formatting of specific data components should remain the same. The advantage of this is that the user should be familiar with the external components formatting while the disadvantage would be that if two external packages use different formats

for a single item, they must either be maintained separately or a conversion algorithm must be written.

- **Easy software distribution.** A stand-alone package has few or none of the external dependencies and as such is simple to install (and uninstall). The drawback of such a package is that it cannot rely on out-of-the-box systems to ease the amount of programming required. This goal should not prevent external components from being used as long as they may be distributed and installed as part of the GrailKnights package.

- **Portable.** The GrailKnights package should be independent of the operating system it is running on. This is an extension of the easy software distribution goal.

We, firstly, need to do the macroscopic design of the package. The package will have to perform three separate tasks: visualisation, data manipulation, and data storage. These three tasks are dependent on one another in that any data flow between the storage and visualisation tasks, may be edited by the data manipulation task. A monolithic package that handles all three tasks is inadvisable as it conflicts with the modularity goal set above as well as being needlessly complex, hindering modifiability and not allowing for easy reusability of components. Further complications arise when one notes that the data manipulation task includes processing large batches of automata and storing their results. This means that the manipulation component and storage component will require more processing power than the visualisation component while running the batches. This leads to the conclusion that the manipulation and storage components should have the option of being distributed onto a different machine to allow improved performance.

In order to design a modular package, we turn to architectural patterns [4]. Architectural patterns offer well-established solutions to architectural problems and describe the attributes of a software system. The architecture which matches the tasks mentioned above, is the 'client-server' architecture [33]. This architecture defines two separate components: clients and servers. The client requests services from the server. Since servers may also be clients to other servers, a 'n-tier-architecture' may be built from multiple client-server relationships. The prominently used 3-tier-architecture is an example of such an architecture. It is made up of the following:

- client tier, most commonly the user interface,

- logic tier, implementing the application or business logic, and

- backend tier, most notably used for data storage.

The disadvantages of a client-server architecture are similar to those of a distributed system (mentioned above).

There are few architectural patterns that provide solutions for GrailKnights at a macroscopic level. A peer-to-peer architecture (in which each peer is considered both client and server) does not lend itself to the goals of GrailKnights any more

than a client-server architecture while it increases the complexity of the system. A peer-to-peer system would make sense for a group of users sharing and comparing information, however, this is not the goal of the current GrailKnights. A layered architecture, in which main components are separated into layers with distinct interfaces between adjacent layers, could be considered to be similar to a client-server architecture at a macroscopic level (with the database on the lowest level and visualisation on the topmost level) except that the client-server architecture is more descriptive. Other architectures' granularity is too small or they do not comply with the goals mentioned above. In the client-server architecture there are many trade-offs that need to be considered:

- **Distribution vs. performance.** Many servers may share the workload equally and in so doing increase overall performance. However, with this increase of distribution, comes a network overhead as well as an increase in the complexity of the system. Overuse of the network may lead to network bottlenecks. A centralised system, while less robust, can more reliably define its performance for a given system configuration.
- **Location of processing.** Batch processing should to be done close to the data storage while transaction processing should be done close to the input/output devices.
- **Distribution vs. security.** The more distributed the system is the more points of attack there are on the system, though this point is less valid as security in GrailKnights is not a goal.
- **Distribution vs. consistency.** A single centralised database reduces consistency problems compared to a distributed system with multiple data storage locations.
- **Software distribution cost.** This measures how difficult it is to distribute, configure and install the system to a multitude of users. The lowest cost will be in a centralised system.
- **Reusability vs. performance vs. complexity.** Placing functionality in the server encourages code reuse while decreasing the size of the client application. The server should be able to handle multiple client requests.

There are also a few distribution patterns that may be considered for a client-server architecture. That is, which roles are assigned to the client and which to the server from the following: presentation, dialog control, application kernel (which contains the application logic), database access, and finally, the database itself.

- **Distributed presentation.** One part of the presentation component is packed in the client while the other part (as well as the remaining roles) are part of the server. This leads to a very thin client which may be easily distributed.
- **Remote user interface.** The whole user interface is distributed and connects to the application kernel on the server.

- **Distributed application kernel.** The application kernel is split between the client and the server.
- **Remote database.** The database exists on its own server and the client defines the database access.
- **Distributed database.** The database is made of different database components so that some may be stored closer to the location where the data may be processed or can allow integration from different database systems.

An example of a 3-tier-architecture would be to use two of the distribution patterns mentioned above, though usually a remote or distributed database with one of the remaining three.

One approach that GrailKnights could have taken is to use the distributed presentation pattern. An example of this is that the client is an HTML-browser and the server a centralised HTML-server. The server would handle the logical flow between web-pages and run the relevant processes while the client would display web-pages and relay web-page changes to the server. The advantage of this approach is that the client would not have to be written as any existing application capable of web-browsing would do. The server part that handles the web pages would not have to be written as many such servers already exist and may be used off the shelf. Location of processing is not an issue since all server processing is done in a localised position. Large amounts of graphical presentation primitives may cause bottlenecks in the network. Data inconsistency may arise between displaying the data and the data maintained in the database, however this is usually for such short periods of time that it may be ignored. Software distribution costs are minimal. This system does not produce readily reusable application components. This design is very suitable for GrailKnights but was not used because of the extra overhead of installing an HTML-server (and all its dependencies) as well as the fact that graphical interaction plays a large part in the GrailKnights package.

The remote user interface pattern may be more complicated than the distributed presentation pattern above in that the programmer must address communication problems that were solved for you by using an HTML-browser and HTML-server. However this allows the interface to be more complex to suit the application's needs. Batches will run smoothly as the server is closer to the data. The performance should be better than the distributed presentation pattern simply because less (but more applicable) data is sent across the network. The software distribution cost is higher because the client is specific to the server. This pattern also allows for different application interfaces to connect to the same server.

A distributed application kernel is a flexible architecture well suited to complex, highly interactive applications. A distributed application kernel implies that the application logic is split between the client and the server. Choosing where this split occurs is difficult. This architecture is the most difficult to design and implement correctly and as such was not considered for GrailKnights.

The remote database architecture has increased network traffic compared to the remote user interface but it is reduced compared to the distributed presentation architecture. In a pure remote database architecture, data needed for batch processing must be sent to the client machine, however, most clients have neither the support nor the input/output needed to process the data. The architecture does not promote reuse of application functionality but data can be shared by different client applications.

A distributed database architecture is where the client-server cut occurs within the database component. The data is distributed across several databases that may be located on different physical machines. The database access level is usually some form of database management system which strives to make the multiple databases appear as one logical database. This architecture was not considered for GrailKnights due to its added complexity which does not forward the goals of GrailKnights.

A 3-tier-architecture using a remote user interface as well as a remote database was therefore chosen for GrailKnights. The use of this client-server architecture does not prevent the client and server from being run on the same computer. Finally, the most important advantage of the client-server architecture lies in its modularity: the client contains all the visualisation mechanisms, while the server is dedicated to maintaining operations requested by the client. The server portion of the GrailKnights project is called the Perceval server, while the client portion of the project is called the Galahad client[1] (see Fig. 3.1). There Perceval becomes the client to a database which will be decided in the following implementation chapter.

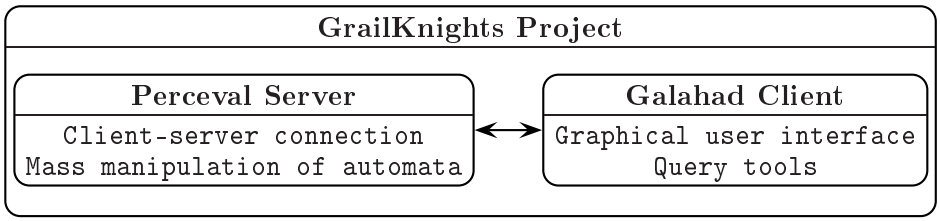

Figure 3.1: GrailKnights overview

[1]The names for the server and client, Perceval and Galahad, were chosen due to their appearances in writings about the Holy Grail. In older writings, Sir Perceval was the first knight to find the holy chalice, while in later versions it was Sir Galahad [1].

3.1 The Perceval server

A layered architecture is well suited for the components of the Perceval server. The components are divided into five modules which are called 'handlers'. This is because each module handles a different component of the server (see Fig. 3.2). These handlers are:

- the client-server connection handler,
- the mass-manipulation of automata handler,
- the database handler,
- the automaton handler, and
- the metadata handler.

The handlers can be seen as separate layers reflecting the flow of data. The topmost layer contains the client-server connection layer, the middle layer contains the mass-manipulation of automata handler and the final bottommost layer contains the remaining three handlers. The client-server connection handler, as its name implies,

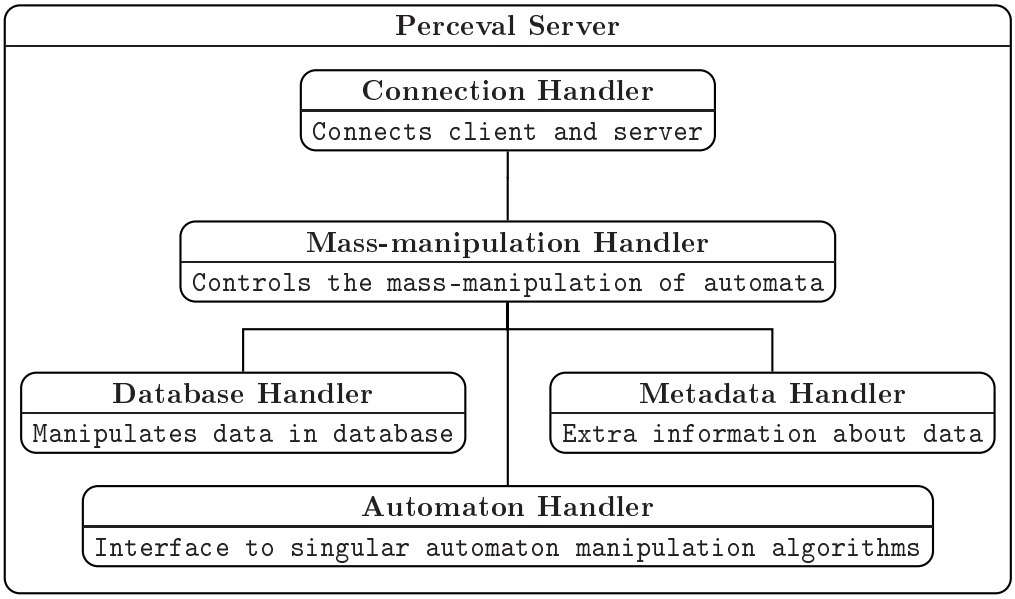

Figure 3.2: Perceval server overview

controls the information sent between the client and the server. The advantages of using a client-server model have already been mentioned and so will not be discussed any further here.

The mass-manipulation of automata handler is the reason for the creation of this project, that is, to manipulate many automata at once. In essence, this module handles requests from clients and obtains the requested answers from the other modules.

The database handler manages a relational database, as well as handling SQL queries and returning their results. The reason for using a database is twofold: keeping track of a large amount of automata is simplified through the use of a database, and the complexity of storing vast amounts of automata on secondary memory is easily facilitated. It is not uncommon for a researcher, while using packages such as Grail+, to create vast numbers of small automaton files. In the extreme case, the problem arises that the operating system's file allocation table cannot handle the (literally) millions of files, possibly resulting in an operating system crash.

The automaton handler manages the singular automaton manipulation algorithms by executing the desired algorithm and returning the resultant value. The automaton handler itself is merely an interface for various implementations of automata manipulation algorithms. The algorithm executed may be external to the GrailKnights project, which implies that a certain amount of time may elapse before a result is ready. Such a time lag is especially likely with large automata or complicated (or time intensive) manipulation algorithms. The program should be able to take this time lag between input and result into account.

There are multiple types of manipulation algorithms with which the automaton handler must contend. The most common types are:

- those with single input and single output, such as minimising an automaton;
- those with two inputs and a single output, such as an isomorphism test and
- those with a single input and multiple outputs, such as deriving multiple statistics from a single automaton.

The inputs are usually the finite automata or regular expressions while the outputs vary between booleans for comparative testing, numbers and strings for statistics and automata or regular expressions for manipulation algorithms.

Finally, the metadata[2] handler indicates to the database handler how the tables and the columns in the tables should be created and managed. The metadata also describes how entire rows of data in a table may be propagated by entering only a single input and then applying automata manipulation algorithms to that input to obtain the data for subsequent columns in the table. The metadata should take the form of scriptable user commands, that is, the metadata should be easily editable by the user in order to facilitate the creation of tables in the database to suit the user's specific needs.

While this handler may seem to violate one of the design principles (namely, not to complicate the task), it is essential to fulfil one of our goals, that is, to require minimal programming by the user. The benefits of the complexity gained by the user having to script a few lines far outweigh the user having to code and recompile a program.

[2]Metadata refers to data that describes other data [3].

3.2 The Galahad client

The Galahad client is the graphical user interface (GUI) for the GrailKnights project. It supplies the user with query tools and visualisation of results from a query. The query tools invoke the underlying mass manipulation algorithms found in the Perceval server as well as general manipulation of the tables in the database. The following are examples of such tasks:

- filling a table with randomly generated automata,
- filling a table with all automata with a specific property,
- applying algorithms to inputs automatically, as declared in the metadata,
- applying a specific manipulation algorithm to all the items in a table,
- showing a table,
- clearing a table and
- plotting relationships between columns in chosen tables.

The graphical user interface design (see Fig. 3.3, page 21) reflects the look and feel of windowed applications currently available for the Windows and Unix operating systems. It contains a menu bar with such menus as main, edit, styles, logging, options and help. The main menu allows the connecting and disconnecting from the server, saving results and loading results and exiting the application. The edit menu reflects tasks available on the toolbar (see below). The styles menu allows for the changing of the look and feel of the application. The logging menu changes the amount of logging information received from the application. The options menu allows the user to change preferences such as configuration information and Perceval server metadata. Finally the help menu provides the help function.

The toolbar contains the tasks for creating automata, placing those automata into a database, applying a specific manipulation algorithm to all the items in a table and placing the results into another table, retrieving a table from the database, clearing a table from the database and plotting relationships between specified data values. These tasks are repeated in the edit menu as per the guidelines given by Johnson [22].

The work screen is divided into five areas: central, top, bottom, left and right. The main central area contains any retrieved information either in table or graph form. The area to the right of the central area contains the user history panel. The history panel contains all the tasks performed in the past on the client program. These tasks can be rerun or deleted in the future as the user sees fit. The bottom area, by default, contains the textual logging output. The amount of text displayed here is regulated by the logging menu in the menu bar. The top and left areas are left unused by default. The panels appearing in all non-central positions may change to any of the other non-central positions, be made to occupy separate windows or minimized as buttons on the bottom area. This functionality is added so that the user remains in control of the work space and so that other panels may be created

at a later stage to be added with ease into the current application.

The simplest feedback the user may obtain is that of simple queries to the database and their resultant tables. Indeed the user may also alter the contents of the tables to suit the needs of the experiments. The user may request to plot a set of values along the x- and another set of values along the y-axis of a graph which is then displayed in the central area. The user may then analyse the graph. Due to the modularity of the design, the plots could be extended into multiple dimensions or otherwise as the user sees fit.

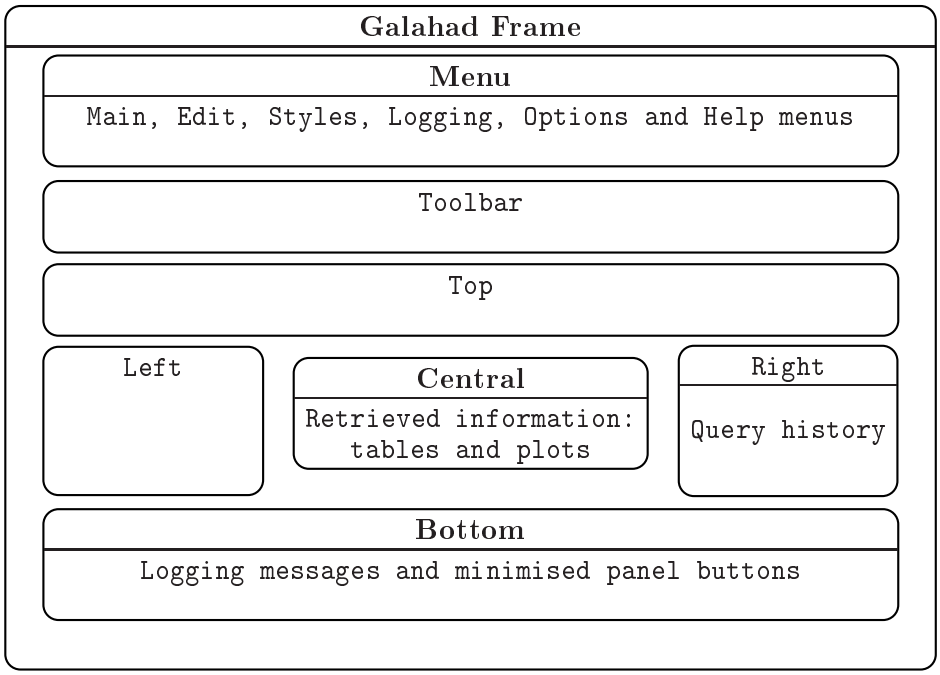

Figure 3.3: Galahad layout design

3.3 Visualisation

Most of the automata packages discussed in Section 2.1 have the ability to display an automaton state graph. Some allow for the entering of input that then visually moves through the state graph, highlighting the relevant states, before the automaton finally accepts or rejects the input. GrailKnights should be able to display an automaton state graph but this is purely for the sake of visually identifying a given automaton. There are various ways to lay out a given automaton state graph. A force-directed graph layout [10] is used. This layout is applied to the states where the states act as a repelling force for all other states and the edges between the states act as springs drawing them together. GrailKnights also allows the user to interactively move the states around with the mouse as well as zooming in and out and panning.

The emphasis with GrailKnights is, however, on the mass manipulation of automata and the visualisation of the relationships between the said automata. To this end, one requires an assortment of graphs and plots. A dual-axis graph may be used to compare numerical values. Such a graph should be able to zoom in for a better view of specific elements in the graph. Scatterplots, with each point representing an automaton, may be used to compare masses of automata depending on how one colours the automata and where one places the automata. One such layout could be described as a force-directed scatterplot. In such a scatterplot, all points are considered to be equally charged particles, which repel one another. Those of same or near values are connected with "springs". This model allows for an ordering of the points in the scatterplot and allows for easy identifying outliers and groups. One example of highlighting in such a layout could be to highlight all those with the same value. It should be noted that the force-directed layout is computationally intensive and as a result, a method of pausing and resuming the computations is an advisable addition.

Using these designs, the next chapter takes these ideas one step further by implementing them.

Chapter 4

GrailKnights: Implementation

This chapter serves as a reference guide to the implementation of the GrailKnights project. We cover the design as discussed in Chapter 3 and highlight the technical issues regarding the implementation.

4.1 Implementation environment

Operating system independence is a desirable feature of a project of this kind and, as indicated in Chapter 3, we follow an object orientated design. Hence, the Java [15] programming language was the obvious choice for an implementation language. The disadvantage of an interpreter such as Java is that it may be slower while executing than similar programs written in a compiled language such as C. The advantage is that it is a highly portable language and as such may run on most operating systems without having to be recompiled. We implemented and tested this first prototype of GrailKnights under Windows XP but foresee no difficulty in it running on Linux.

Note that the communication interface between the Perceval server and Galahad client was implemented using Remote Method Invocation (RMI) [11]. RMI is a Java dependent communication protocol and as such is usable on operating systems that use Java. This is discussed further in Section 4.2.

The database used to store the myriad of automata and related information is IBM Cloudscape [21] which is a Java based database and, as such, retains all functionality in both Windows and Linux operating systems. The choice of database is discussed in Section 4.3. The language used to reference the database directly is a form of SQL, whose basic commands are the same as most implementations of SQL. This allows for any database that uses SQL to replace the IBM Cloudscape database. Further information on the data and metadata is given in Section 4.4.

The GrailKnights prototype uses Grail+ [31] to manipulate single automata and regular expressions. Grail+ is written in the C++ programming language which has the problem that its executables must be compiled to be either Windows or Linux compatible, not both. The GrailKnights prototype that runs on Windows uses Grail+ executables compiled for Windows. A Linux executable of Grail+ exists

and should simply replace the Windows executables when installing GrailKnights on Linux.

4.2 Communication

The communication interface used between the Perceval server and the Galahad client is known as Remote Method Invocation (RMI) [11]. RMI, as the name implies, allows the client to access procedures on a remote machine as if those procedures were on the local machine. RMI was chosen because of this transparency, as well as its simplicity.

Alternatives to RMI are invariably more complex. For example, Java does support the use of sockets. The advantage of sockets is that they are flexible and generally sufficient for communication. The disadvantage of sockets is that the client and server require application-level protocols that encode and decode messages for exchange.

4.3 Database

Java is compatible with most relational databases due to the many types of the JDBC (Java database connectivity) drivers available:

- **Type 1, JDBC-ODBC bridge.** This can be used to connect to any database that supports the ODBC (Open database connectivity) driver. This type entails an overhead as the driver converts each JDBC method call to an ODBC method call. This type is only used if none of the other types are available. The driver must be available on the client side as this is where the conversion is done. In some cases (for example applets) the driver may not be available and therefore this driver type cannot be used.

- **Type 2, Native-API Driver.** This driver is database specific and converts the JDBC methods directly to native calls of the database. The driver must be available on the client side with the same disadvantages as for type 1. This has an advantage over type 1 drivers in that it skips the translation to and from the ODBC protocol.

- **Type 3, Network-Protocol Driver** or **Pure Java Driver for Database Middleware.** This makes use of middleware between the database and client which converts the JDBC methods directly (or indirectly) into native database methods. The advantage is that there are no client side drivers needed and a single middleware may service multiple different databases. The disadvantages are that the middleware may be result in a performance bottleneck and the middleware is required to have database specific coding.

- **Type 4, Native-Protocol Driver** or **Direct to Database Pure Java Driver.** This driver converts the JDBC methods directly to native calls of the database. The driver is written in Java and runs on the clients JVM

(Java virtual machine) and as such is platform-independent. This driver has a marked performance improvement compared to the other driver types. The disadvantage is that each database must have its own driver written for it.

Cloudscape is a Java-based database and as such is platform-independent and can be easily distributed as a single jar file. This implies that Cloudscape does not require its own installation. The JDBC driver that Cloudscape uses is type 4 and therefore has a higher performance than other types. Cloudscape can be 'embedded' inside Perceval's JVM and does not require an external process of its own. This decreases network traffic and turns the GrailKnights package from what was a 3-tier-architecture, into a simpler 2-tier-architecture. It should also be noted that this does not hinder the Perceval server's ability to connect to an external database if necessary. A disadvantage of using Cloudscape is that it is a relatively new technology compared to some databases and as such may contain bugs that have not yet been solved. Another disadvantage is the licensing for Cloudscape: it may be used for free for academic purposes but may not be distributed and the user must register at IBM's site in order to download it. It may be preferable to use an existing database, if it has already been installed on the system. As was mentioned earlier, any database that can be accessed through JDBC is a valid alternative for the GrailKnights package.

```
<table name="fm" key="fm_id">
  <col name="fm_id" type="INT" id="notnull"/>
  <col name="fm" type="CLOB" input="true" gtype="fm"/>
  <col name="min_dfa" type="CLOB" filter="fmdeterm fmmin"
   gtype="fm"/>
  <mcol number="1" type="INT" filter="fmstats"
   from="min_dfa" filterIgnore="1 2 4 5">
    states_total
  </mcol>
</table>
```

Figure 4.1: Example of metadata extract showing table 'fm'

4.4 Data and metadata

The automata and their associated data are stored in the database. One way to define the tables of the database (and their intercolumn relationships) without having the user write and compile code, is to create a metadata file. This metadata file is an XML[1] file which has the advantages of being easy to read by humans as well as being easily parsable by the computer. The XML file requires validation to ensure its correctness. The validation file used is known as a schema file and in this case, it is the *meta.xsd* file. The *meta.xml* file not only defines the table structures in the database but also describes the valid automata manipulation algorithms as well.

[1] Extensible Markup Language [17]

tab3

FM_ID	FM	MIN_DFA	STATES_TOTAL
444	FM	FM	3
446	FM	FM	5
447	FM	FM	1
448	FM	FM	4
449	FM	FM	5
451	FM	FM	2
452	FM	FM	2
453	FM	FM	5
454	FM	FM	4
455	FM	FM	6
456	FM	FM	3
457	FM	FM	5
459	FM	FM	2

Figure 4.2: Galahad client view of populated table 'fm'

Fig. B.1 (page 63) is an example of a metadata file.

The tables and columns in the database are created to match the information given in the metadata file. Fig. 4.1 (page 25) is an example of a possible extract from the metadata file. The table's name is 'fm' while its 4 columns are 'fm_id', 'fm', 'min_dfa' and 'states_total' as can be seen from the Galahad client view of the same table in Fig. 4.2. In this way the user can define the structure of the tables in the database to suit his/her needs. The metadata file requires a single input column to which, in this case, the original input will be sent. There after, any column that has the filter attribute defined in the metadata will be filled with the result of that filter using the input column as input. In this example, if a NFA is inserted into the table, it will firstly be placed in the 'fm' column. Secondly, the 'min_dfa' column will be filled with the minimised DFA version of the input NFA. Finally the 'states_total' column will be filled with a number indicating the number of states from the 'min_dfa' column (the from attribute shows which column to use as input. If the from attribute does not exist, the input column is used). Note that the automata are all in Grail+ format. The number of rows in a table depends on the number of items the user wishes to place in that table. In this example, if the user generates a million NFAs, the table will contain a million rows. This satisfies the transparency goal laid out during the design phase. For further information on building the metadata file see Appendix B.

The parser that is used to parse the metadata XML document is known as a SAX (Simple API for XML, where API stands for Application Programming Interface) parser. The SAX parser is a simple and fast parser. The other parser used by the Galahad client to parse the configuration file is known as a DOM (Document Object Model) parser. The DOM parser is slower but has the benefit of creating a tree structure which can be edited easily.

4.5 Perceval server

The Perceval server is a modular package, as discussed in Chapter 3, which consists mainly of the client-server connection handler, mass-manipulation of automata handler, database handler, singular automaton handler and metadata handler (see Fig. 4.3). Additionally, filling lesser roles are the configuration handler, message logging mechanism and other useful tools (see Section 4.5.8).

The database handler and the automata manipulation handler facilitate the communication between the database and the automata package. The metadata handler and configuration handler both use a file in XML format as basis for their tasks. The logging mechanism is added to ease debugging and provides the user with useful feedback.

Most of the classes in the Perceval server use a Java interface which is a realisation of an abstract data type (ADT, not to be confused with an abstract Java class). An ADT shows *what* each operation does but not *how* it is done. That is, an interface is simply a list of method declarations. A Java class that specifies *how* all the methods are performed in a given interface, is said to *implement* that interface. For a single interface, many implementing classes may be written, each with different ways of doing the same operation specified by the interface.

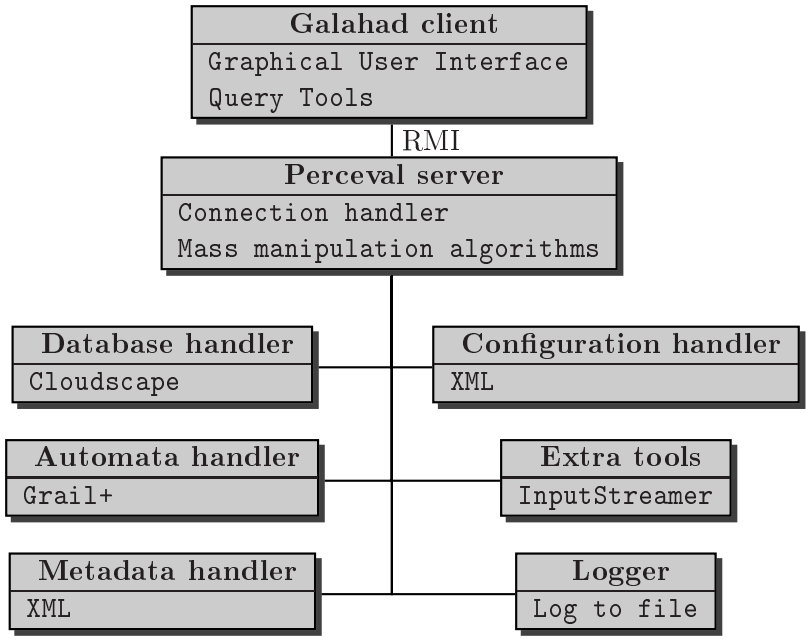

Figure 4.3: Perceval server implementation

4.5.1 Perceval RMI

The connection and mass-manipulation handlers are combined into a single class known as the PERCEVAL class. The PERCEVAL class handles the connecting client and any requests the client may perform. The merger of these two handlers into a single class defeats the purpose of the modularity of the server. However, for the alpha version, the pace at which a complete system can be reached is greatly increased and the splitting of the functions can be done with ease at a later date.

The PERCEVAL class can be seen as the central program, which uses all the other classes to complete its tasks. We shall discuss this class in more detail once the other classes have been discussed. The portion that may be discussed at present is the RMI interface portion of this class. The RMI interface is known as the KNIGHTI interface. The client uses the KNIGHTI interface to remotely access the methods on the PERCEVAL class which implements the KNIGHTI interface. A code extract of the KNIGHTI interface is shown in Fig. 4.4. For the KNIGHTI interface to be a valid RMI interface, it must extend the interface JAVA.RMI.REMOTE and all its methods should throw a JAVA.RMI.REMOTEEXCEPTION.

```
public interface KnightI extends java.rmi.Remote {
  public MetaTable getMetaData() throws RemoteException;
  public KnightXYPlot getXYPlot(String x,String y) throws
   RemoteException;
  public KnightTable getTable(String tableName) throws RemoteException;
  public KnightTable getTableWithColumns(String tableName,
   String[] cols) throws RemoteException;
  public KnightTable applyTwoColFilter(String filter, String src_col1,
   String src_col2, boolean returnResult, boolean saveTable,
   String dest_col1, String dest_col2) throws RemoteException;
  public KnightTable applyDirectQuery(String query) throws
   RemoteException;
  public void applyDirectUpdate(String update) throws RemoteException;
  public void clearTable(String tableName, boolean inMeta) throws
   RemoteException;
  public void setTableName(String tableName) throws RemoteException;
  public void insertItem(String item) throws RemoteException;
  public void shutdown() throws RemoteException;
  ...
}
```

Figure 4.4: KnightI interface code extract

4.5.2 Automata handler: Grail+

The default automata manipulation algorithms package used is Grail+ [31]. Grail+ is a C++ package. The automata handler namespace contains the AUTOMATON-

HandlerI, GrailHandler and StreamTracker classes. The automata handler uses the LogBase, Logger and Config classes. The AutomatonHandlerI interface describes the methods available to the Perceval Server. The *filter* method requires two strings as input. The first string is a *filter-chain* description consisting of the names of filters (separated by a space). The order of execution is always from left to right. The second string is the item required for the first filter in the chain.

The GrailHandler class is an implementation of the AutomatonHandlerI interface. This implementation is designed for the use of Grail+ automata manipulation algorithms called 'filters'. The two primary methods in the GrailHandler class are *runProcFromFile* and *runProc*. Both methods attempt to execute the command defined by the input string *procName*. The *runProc* method requires a single string which is streamed directly into the executed process. The *runProcFromFile* method requires two strings which are first written to temporary files and then defined as part of the executing command. Both these methods make use of separate threads, in the form of the StreamTracker class, to store any output from the executing process. The StreamTracker class provides a convenience method to check that all information from the executing process has been stored. This is due to the fact that in some cases the process itself may complete before the process's output has been stored. If an error occurs (for example from a process executing incorrectly) the stack trace is dumped to the log file and a null result is returned.

To use a different automata package, one would have to write ones own class that implements the AutomataHanlderI interface. Then one would need to edit the configuration file to use that class instead of the GrailHandler class by adding the following: `<handler name="myHandler" value="javapath.MyHandler"/>`. Finally one needs to edit the metadata file and add the filters found in the new handler. It should be noted that the current implementation of the GrailHandler class handles a select group of Grail+ algorithms as proof of concept.

4.5.3 Database handler: Cloudscape

The default database handler uses IBM Cloudscape [21]. The database handler namespace contains the DBHandlerI, CloudNine and DBException classes. The database handler uses the LogBase and Config classes. The DBHandlerI interface defines all the methods required in order for the database handler to function. One must always call *connect* to start up the database. This is followed by specifying which database to use. If the specified database does not exist it must be created. Before and after each group of database altering methods (create/insert/update/drop) one should call the *startTransaction* and the *endTransaction* methods respectively. If anything goes awry during a call to the database, a DBException is thrown.

The DBHandlerI interface is implemented by the CloudNine class. The CloudNine class is designed for use with the Cloudscape database. The CloudNine classes connect method starts an embedded Cloudscape database by default. The Cloudscape base directory is defined in the configuration file.

In order to connect to a different database a new handler must be written that complies with the DBHandlerI interface and the configuration file must be updated to use the correct handler.

4.5.4 Metadata handler

The metadata handler namespace contains the Meta, MetaHandler and MetaParserI classes. The metadata handler uses both the Logger and Config classes. The Meta class is a singleton (static class) that allows easy access to metadata required for various database operations by storing the information in its data structures. The metadata itself is obtained by parsing an XML file, defined in the configuration file.

The MetaParserI interface defines the methods required for use in the Meta class. These methods include the parsing of the specified file, the retrieval of the data structure and the retrieval of the database name to be used. The MetaParserI interface is implemented by the MetaHandler class. The MetaHandler class is a SAX parser that parses XML files.

4.5.5 Algorithms of interest

One of the more complex problems is the batching of an algorithm that has two inputs. For such a batching we define two input sets, A and B, which may both refer to the same set. The algorithm must use as input all combinations of elements from set A and set B, where elements of set A are always the first input and elements of set B are always the second. In the case where the results change if the order of inputs change, the batch may be run again with the sets exchanged. Such a batch is expensive because of its $O(n^2)$ where n is the number of algorithm executions. Because of this polynomial running time a batch could take a long time complete with large input sets. For example, if set A and set B both contained 1000 elements, then n would be 1 000 000. To make such batching viable, two modifications are made.

- Firstly, if sets A and B are both the same set, then it is assured that no two items that have already been used as inputs are used again. That is to say, the first item is used as input $n-1$ times (it is not used with itself), the second $n-2$ times, the second last twice and the last item is used only once, where n is the number of elements in the input set. This alteration drastically decreases the number of algorithm executions required. For example, if the input set contained 1000 elements, the algorithm would be run 500 000 times, halving the number of executions required.

- If A and B are the same set, we also assume commutativity, that is, $algorithm(a, b)$ has the same result as $algorithm(b, a)$ where a is an element of set A and b is an element of set B. Once all the results are found for a specific element, we may remove all the elements (from the sets) used to obtain these results from further algorithm executions as we already have all the results

required. In the case where the algorithm is not commutative, a copy of the set may be made to a different physical location and so obtain two sets.

The reader may note that the running time of some experiments may take days and there is no guarantee that the server may remain active for the entire period of the experiment. The server includes a method of maintaining the state of the experiment by saving the completed portion of the experiment to temporary tables in the database. Each experiment is given a unique identifier and a special table (not part of the metadata) designates which temporary table that experiment uses.

An example of an algorithm with two inputs is the Grail+ isomorphism testing algorithm. Isomorphism testing is commutative and so the modifications mentioned above are used to decrease processing time. Note that in the case of isomorphism testing, the user should ensure that the automata being tested are minimised DFAs.

4.5.6 Configuration handler

The namespace perceval.util.config contains the Config, ConfigHandler and ConfigParserI classes. The configuration class Config is a singleton (static class) that allows easy access to properties defined in the configuration file. When the Perceval server is started, the Config class attempts to populate itself using the method defined in the ConfigParserI interface. The default implementation of the ConfigParserI interface is the ConfigHandler class. The ConfigHandler class is a SAX parser used to parse the XML file provided according to its own validating schema (*config.xsd*). A discussion of the configuration file is given in Appendix C.

4.5.7 Logging mechanism

The perceval.util.logger namespace contains the LogBase and Logger classes. The Logger class requires the Config class for configuration purposes. The Logger class is a singleton (static class) that allows easy logging to a file specified in the configuration file. If this log file already exists, any additional information is appended. Otherwise the file is created. If the file specified is invalid, any logging will be streamed to the standard Java output.

The LogBase class forms a base from which other classes can be extended in order to ease the use of the singleton Logger class. This is done by using debug methods in the subclasses of LogBase. Plainly, classes that are already extended cannot benefit from this abstraction (nor can static classes) and so may use the Logger class directly.

4.5.8 Additional tools: InputStreamer

To expedite the addition of data to the database, the Perceval server includes tools that act as clients. An example of such a tool is the InputStreamer, which allows the user to input multiple items directly into a specific table in the database depending on how the metadata file is configured. The tools namespace contains the InputStreamer class. Because the tools are in effect specialized clients of the

Perceval server, all tools require the use of the CONFIG and KNIGHTI classes. All classes in this namespace are tools which allow for the automation of a specific action that would be tedious to perform by hand or through the graphical user interface. The INPUTSTREAMER class requires a configuration file similar to the one used by the PERCEVAL class (the same one can be used, because extra information in the XML file is parsed but not used). There are two ways to use the INPUTSTREAMER class. The first is to run it with a configuration file as input, just as one would run the Perceval server. The INPUTSTREAMER would then display a command prompt allowing the user to type in an item to be placed in a database table (which is specified in the configuration file). The item is ended by typing a separator character (or string) on its own line in the prompt. The second method is to start the INPUTSTREAMER class in the same way but to pipe a file or program output (with separator characters included) to the INPUTSTREAMER program. This serves as an automated way of inserting many items into the database with a single command. The item is then placed in the input column of the specified table and any filters for that table are applied to that input as defined by the *meta.xml* file in the Perceval server.

4.5.9 Adding different handlers

The Perceval server uses the automata handler, database handler and metadata handler to perform the requests it receives from the client via the RMI connection. These handlers each use a specific interface. The automata handler uses the AUTOMATONHANDLERI, the database handler uses the DBHANDLERI interface and the metadata handler uses the METAPARSERI interface. These interfaces may be used to create handlers that add additional functionality to the GrailKnights package. For example the GRAILHANDLER class implements the AUTOMATONHANDLERI interface so that GrailKnights may have access to Grail+ automaton manipulation algorithms.

4.5.9.1 Changing the database

In most cases, no coding will have to be done to use a different database since Java connects to the database using a JDBC driver. All that is required are some changes to the configuration file (shown in Fig. 4.5, page 33). For a type 4 driver on an embedded database, the only items that may need changing are the 'user', 'password' and 'dbhome'. If a non-embedded database is used the 'framework' should be changed to anything but 'embedded', and the 'driver' and 'protocol' should be changed to match the database being used.

4.5.9.2 Changing the automata manipulation package

In order to add an entirely new automata manipulation package to GrailKnights, a handler that implements the AUTOMATONHANDLERI interface must be created. Then the configuration file must be edited so that the Perceval server will use the new handler. Finally, the metadata file must be edited to include the new automata manipulation algorithms.

As an example, let us assume a C executable named 'atm' exists. This executable has two filters, one called 'fmtore' which converts an FA to a regular expression and another called 'fmreverse' which returns the reverse of the given FA. It is up to the user to ensure correct formatting of the FA. In this case we assume 'atm' uses a similar format to a Grail+. Firstly, we create an AtmHandler class which extends the GrailHandler class. This is to take advantage of the `runProc` method which runs an external program with the given input. In the AtmHandler class we override the `runStringFilter` as shown in Fig. 4.6. We assume the command line for running an atm filter is given by `atm -[filtername] input`. The second step is to update the configuration file by adding the following:
`<handler name="atm" value="perceval.handler.automata.ATMHandler"/>`
Lastly we update the metadata file to reflect the new filters as shown in Fig. 4.7 (page 34).

It should be noted that while the backend maintains transparency in the case of automaton format, the Galahad client currently works exclusively with the Grail+ format. This is a shortcoming that should be addressed in future work.

```
<class className="CloudNine" debug="true">
  <str name="framework" value="embedded"/>
  <str name="driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
  <str name="protocol" value="jdbc:derby:"/>
  <str name="user" value="user1"/>
  <str name="password" value="user1"/>
  <str name="dbhome" value="{dbDir}"/>
</class>
```

Figure 4.5: Extract of the configuration file: Database handler setup

```
public String runStringFilter(String filterName,String[] input) {
 if (filterName.equals("fmToRegEx")) {
   return runProc("atm -fmtore",input[0]);
 else if (filterName.equals("fmReversed")) {
   return runProc("atm -fmreverse",input[0]);
 }
 else return null;
}
```

Figure 4.6: Changing automata manipulation package: Example code extract

```
<filters>
  <filter name="fmToRegEx" inputNum="1" inputType="FM"
    handler="atm" outputNum="1" outputType="RE"/>
  <filter name="fmReversed" inputNum="1" inputType="FM"
    handler="atm" outputNum="1" outputType="FM"/>
</filters>
```

Figure 4.7: Changing automata manipulation package: Metadata filters

4.6 Galahad client

4.6.1 Galahad overview

GalahadUI is a graphical user interface which makes extensive use of Java Swing [12] components to create an interface (shown in Fig. 4.8, page 35) which is independent of the user's operating system. The graphical user interface not only provides tools for the creation of queries, but also provides multiple means of displaying the results of the query in a user-friendly fashion. The many features of the GalahadUI are described in more detail below.

The GalahadUI application is comprised of many different modules that can be altered without affecting the working of other modules. These modules are also described in more detail below.

4.6.2 Queries and results

The querying abilities and result displaying abilities of GalahadUI should be powerful enough to facilitate any likely operation (or group of operations) that a user may wish to apply to a set of automata. Applying an operation should also be intuitive.

4.6.2.1 Entering queries

Queries can be entered in one of two ways. Firstly, the user could enter a database query directly (in SQL in this case). Secondly, the user could have a query created by the 'query creator' which builds a query, step-by-step, according to the user's requirements.

There are two query types: the simpler queries that obtain all information directly from the database and the more advanced queries which apply automata filters to database queries. The simple queries (those that do not make use of automata filters) can benefit from query previews [38] which give an indication as to how many items would be returned if the query was issued. This gives the user some warning as to which queries would return too many items or queries which return no items. Query previews do require preprocessing time in order to calculate the cardinalities of user defined datasets. If query previews prove to be too much of a hindrance for a user, they may be switched off.

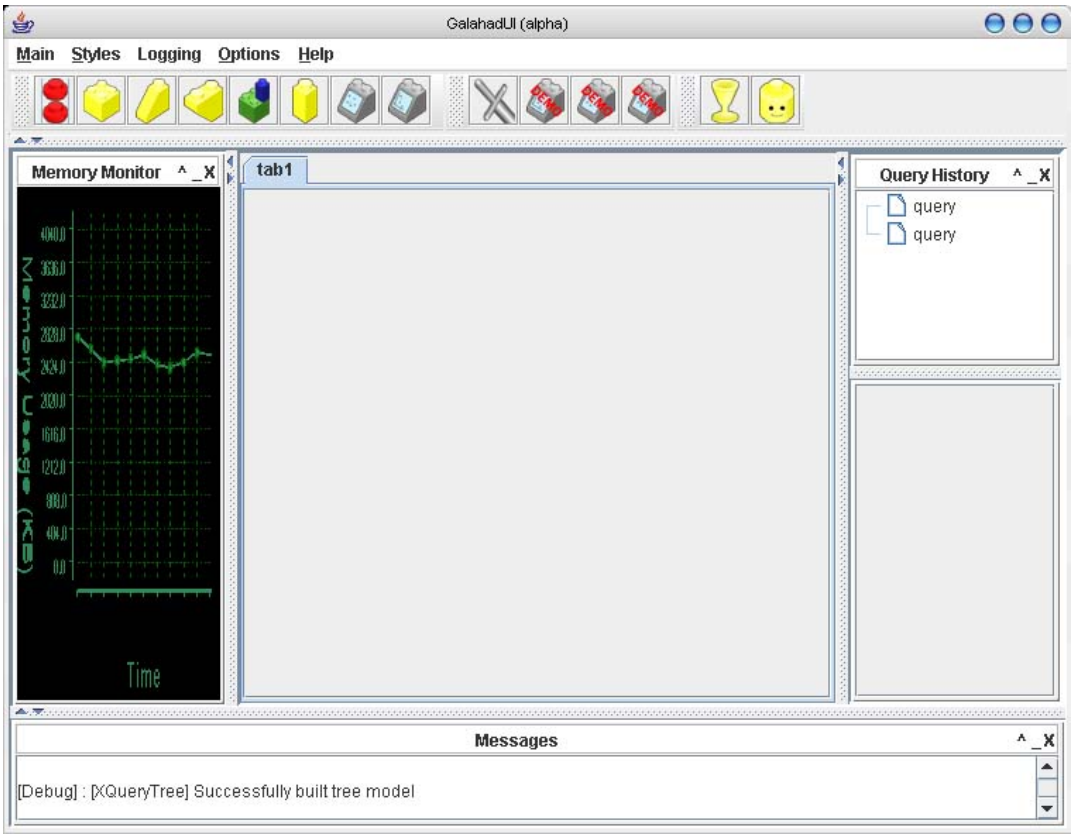

Figure 4.8: Galahad user interface

The advanced queries, as stated before, are queries which have an additional level of complexity in that they allow the addition of automata filters to the normal query. One method that could be used to simplify the use of filters is to have the filters output to the database table and use the simple queries to query that table.

4.6.2.2 Displaying results

The simplest form of displaying results is the table form. The table form, however, does not allow for quick intuitive conclusions, especially if the table has many rows with a highly varying number of values. The graphical utilities of GalahadUI allow for the plotting of results into graphs by specifying the graph axes in terms of the tables columns. Graphs are used for plotting numerical values. For non-numerical values, diagrams can be drawn to show a specific relationship (for example, all automata isomorphic to a specific automaton). This product (GrailKnights) includes software developed by the Group for User Interface Research at the University of California at Berkeley, namely the Prefuse package [19]. This package is used and extended to display the graphs and scatterplots seen in GrailKnights.

4.6.2.3 User history

Once a query is executed, it is saved to the history queue. The corresponding results are saved in a cache. This allows for previous queries to be reviewed quickly and run if necessary. The history queue will not keep all results in the cache, since that would require too much memory space. Instead, only the results of the latest query are cached.

4.6.3 Core modules

GalahadUI is written to be modular in nature. Each module uses a module specific interface to describe which method calls can be made to that module. All the modules supply their own methods for displaying their various Swing components. The central module or program core is the GalahadUI itself. The core module loads other modules as specified by the configuration file (*conf.xml*). There are some helper classes that form part of the core. One such class is the GalahadUIAction class and another is the GalahadToolBar class. These two classes simplify the creation of actions and tool bars respectively. The GalahadApplet class allows the GalahadUI application to run as an applet in a web browser.

4.6.3.1 Connection module

The connection module (*KnightConnecter*) facilitates the connection between the Galahad client and the Perceval server. The module is able to attempt to make a connection, keep track of whether a connection is made and disconnect the client from the server.

4.6.3.2 Table module

The table module is responsible for the display and editing of a table of values in the application. The table allows sorting via its headers and the control and shift buttons.

4.6.4 Modules

This section describes the GalahadUI modules that do not form part of the core application but are, nonetheless, essential for its functioning.

4.6.4.1 Internal message module

The internal message module is implemented by the IMCourier class. It is used by all the other modules for debugging purposes. It handles all text messages that are displayed on the console and the status bar. There are five message types:

- **Informative.** These are information messages meant for the users, for example, the feedback message when a dialog is canceled.
- **Warning.** These messages warn the user that a recoverable error has occurred, for example, a nonsensical input in the configuration file.

- **Debugging.** These are informative messages for the programmer. These should be switched off before distribution.
- **Error.** These are error messages meant for the programmer. They are usually unintelligible to the user and should be sent to the programmer.
- **Critical.** These errors prevent the system from functioning and may require the system to be shut down.

The module allows some or all message types to be suppressed via the Logging menu.

4.6.4.2 Language resource module

All the other modules use the language resource module to obtain the correct language string from the resource file (normally *galahad.properties*). The module uses the functionality of the *java.util.ResourceBundle* class to load the resource file. Plainly put, the language resource chosen is dependent on the computer's locale and language settings. If a specific language resource is not found, the default (in this case, English) is used.

4.6.4.3 Styles module

The Styles module allows the changing of visual themes for GalahadUI and can be accessed from the Styles menu. The four styles are: Java (Metal) theme (which is the default), the Motif theme, the Macintosh theme and the Windows theme. The last two are operating system dependent and if chosen will change the GalahadUI application to match the current style of the Macintosh or Windows operating system respectively.

4.6.4.4 DOM XML converter module

This module makes extensive use of the *javax.xml.parsers.** and *javax.xml.validation.** classes to encapsulate the conversion of an XML file to a DOM tree structure [16]. The modules that use this module include their own methods to convert the DOM tree structure into a more readily usable structure.

4.6.4.5 Configuration module

The configuration module (*Config*) uses the DOM XML converter module to create a data structure that retains all the relevant properties described in the configuration file (usually *conf.xml*) for later use. This Config class, while not the same class as the one used in the Perceval server, does follow the same rules and the results are identical.

4.6.4.6 XTree modules

The two XTree classes (namely *XTree* and *XTreeNode*) use the DOM XML converter module to create a visual tree structure from an XML file that may be browsed and edited.

In this chapter we took the design discussed in the previous chapter (Chapter 3) and fleshed it out to create a functional package. In the next chapter we perform automata experiments using the created package and analyse the results.

Chapter 5

Evaluation and Analysis

In this chapter we introduce examples of automata theory problems that typically require visual pattern analysis. Each section includes the following:

- problem statement,
- how the problem may be investigated using GrailKnights,
- the visualisation provided by GrailKnights and
- analysis using the GrailKnights package.
- evaluation of the GrailKnights package.

5.1 Example 1: How many NFAs are succinct?

5.1.1 Problem Statement

Given a number of randomly generated n-state NFAs, how many of the equivalent minimised DFAs have $2^n - 1$ states? That is to say, how many NFAs are succinct?

The expected result is a marked decrease in the number of minimised DFAs with their number of states nearing $2^n - 1$. In fact, there should be many more minimised DFAs with a small number of states than DFAs with a large number of states.

5.1.2 Using GrailKnights

GrailKnights requires the following steps in order to investigate this problem:

- set up the metadata file,
- randomly generate the automata,
- potential adjustments and
- visualisation of the results.

GrailKnights requires the user first to set up a section of the metadata *XML* file as given in Fig. 5.1. This file not only provides GrailKnights with database tables and columns but also the information on which column the input (in this case) NFA will be placed in. It also specifies which Grail+ filters to apply to which column and in which column to place the results. In this case the first column in our database will give a unique identifying number for each entry. The second column contains the input NFA. The third column contains the minimised DFA which is obtained by applying the *fmdeterm* and *fmmin* Grail+ filters to the input NFA given in column two. The final (third) column contains the total number of states in the minimised DFA. Note that the Grail+ filter *fmstats* delivers five outputs of which we are only interested in one: the total number of states.

Once the metadata XML file has been saved, GrailKnights is able to produce randomly generated n-state NFAs through the `Generate Random Finite Automata` dialog (Fig. 5.2, page 41). The dialog requires the table where the items are to be placed (in this case simply "fm") as well as the number of automata to be generated. For this example we use a relatively small number of automata; namely 1000. The main configuration file allows for changing the number of states, number of transition symbols, density of the final states and density of transitions. For the example we use the defaults (4 states, 2 transition symbols, 25% transition density and 50% final state density.) Once the `Ok` button is clicked, the Galahad client will randomly generate 1000 finite automata and send them to the server where they will be stored and any dependent columns will be automatically calculated.

```
<table name="fm" key="fm_id">
  <col name="fm_id" type="INT" id="notnull"/>
  <col name="fm" type="CLOB" input="true" gtype="fm"/>
  <col name="min_dfa" type="CLOB" filter="fmdeterm fmmin"
  gtype="fm"/>
  <mcol number="1" type="INT" filter="fmstats"
   from="min_dfa" filterIgnore="1 2 4 5">
    total_states
  </mcol>
</table>
```

Figure 5.1: Example 1: "fm" table section of metadata *XML* file

To view the table as a whole, one can use the `Table Query` dialog (Fig. 5.3, page 41) which returns the specified table in a sortable table format. Using this, one can see that there are a number of items in the *STATES_FINAL* column that are zero. This indicates that the corresponding NFA contains no final states and as such, their minimised DFA is empty. We remove these by doing a simple SQL query: *DELETE FROM FM WHERE STATES_TOTAL=0* using the `User Defined Query` dialog (Fig. 5.4, page 41).

Figure 5.2: Example 1: Generate automata dialog

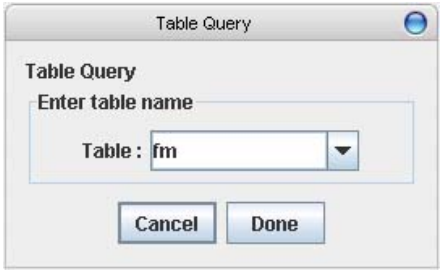

Figure 5.3: Example 1: Table query dialog

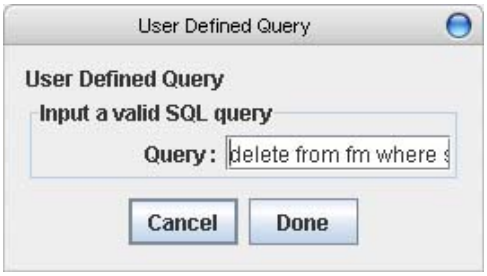

Figure 5.4: Example 1: User defined query dialog

5.1.3 GrailKnights Visualisation

We now view the items in a force-directed scatterplot (discussed in section 3.3, page 21) using the `Create Scatterplot` dialog (Fig. 5.5, page 42). To do this we select the "fm" table and use the "fm_id" column which contains the identifiers for the automata, the "min_dfa" column to represent the automata themselves and finally the "$states\_total$" column which contains the values we are plotting. The second part of the dialog is not used in this example and so we simply click the `Done` button.

As one can see in Fig. 5.6 (page 43), the points sort themselves according to the number of total states. The highlighted points are those that share a value with the one selected by the user in real time.

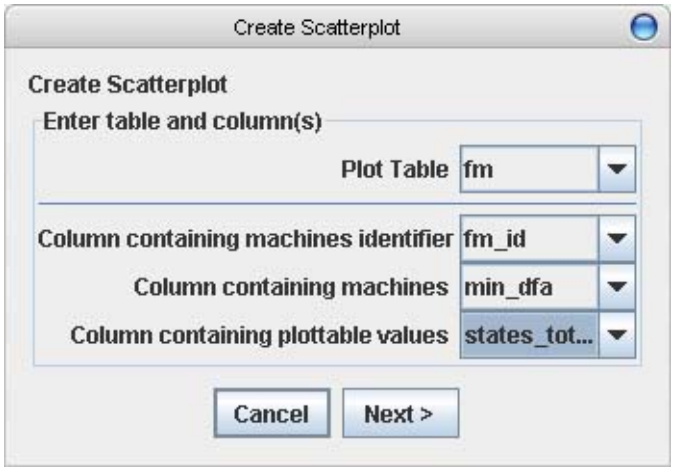

Figure 5.5: Example 1: Create scatterplot dialog, part 1

5.1.4 Analysis

Doing the same experiment without the use of the GrailKnights package, that is to say, using only an automaton package such as Grail+, would require the user, first, to write a script which creates many randomly generated automata. These automata would then have to be saved as a myriad of small text files each containing a single automaton in the Grail+ format. Another script would have to be written to run each of the small files through the three Grail+ functions (*fmdeterm*, *fmmin* and *fmstats*). The script will also have to count all those automata that have the same number of total states and store them. Finally these amounts would have to be graphed using a graphing program. Alternatively, the above two scripts could be combined into a single script that creates the automata, runs through all the above functions and saves only the relevant attribute (in this case, the number of states), counts the number of occurances of each attribute, and finally plots these numbers using a simple plotting program. This has its drawbacks, in that it is impossible to retrieve the original automata for comparison or further experimentation.

Using this method has some problems. The lack of a database means that the number of small files that may be created is dependent on the capacity of the file system of the operating system, which may not be sufficient to handle millions of small text files. The access to the outputs of each intermediate step is limited. For example, one cannot simply view all the automata which provided 14 state minimised DFAs.

Using GrailKnights, however, delivers the expected result, as one can see from Fig. 5.6 (page 43); as one nears 15 states the number of automata dramatically decreases. For the relatively low number (1000) of randomly generated NFAs, it is expected that there are a very small number of 15 state minimised DFAs (zero is most likely) while there is only one 14 state minimised DFA. The fact that there are no 13 state DFAs can be attributed to the relatively low number of automata

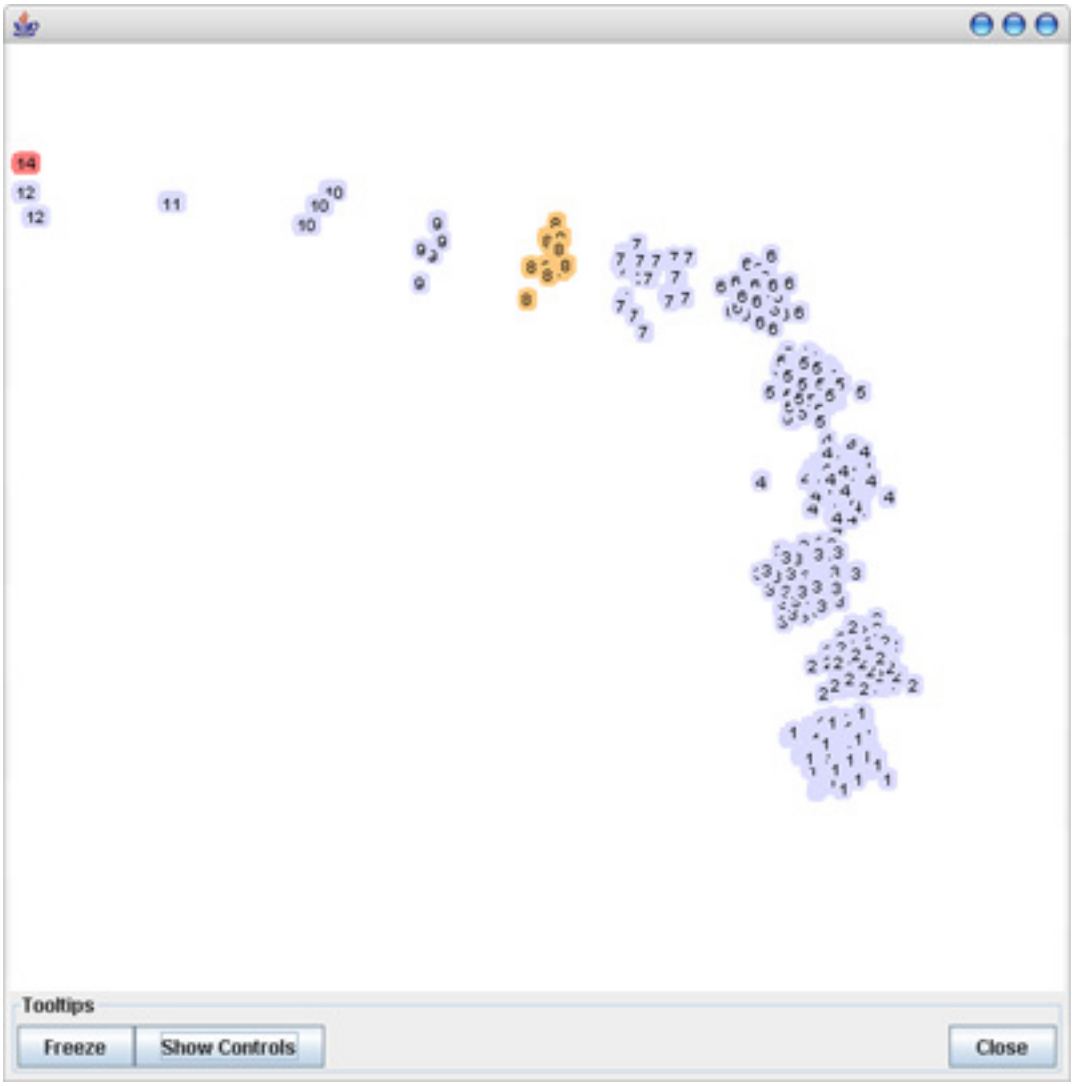

Figure 5.6: Example 1: Scatterplot indicating number of automata with n states

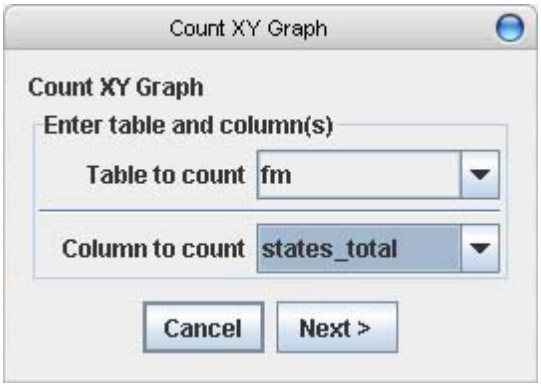

Figure 5.7: Example 1: Plot graph dialog, part 1

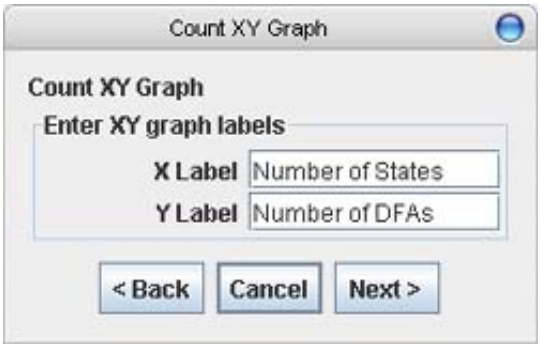

Figure 5.8: Example 1: Plot graph dialog, part 2

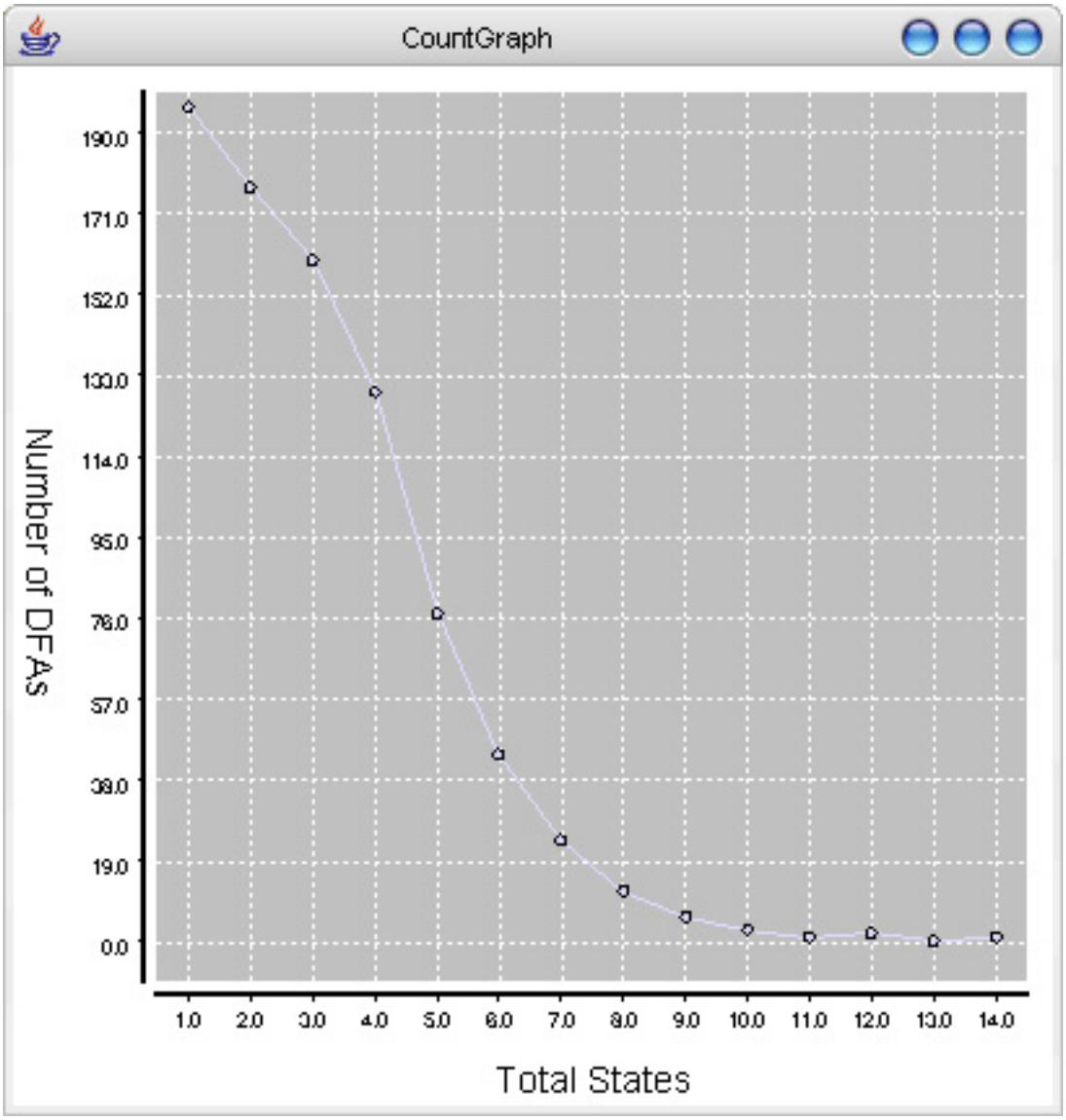

Figure 5.9: Example 1: Graph of number of n-state minimised DFAs over n

generated; the same explanation holds for the two 12 and single 11 state minimised DFAs. From the 10 state through to the 1 state minimised DFAs, one can clearly see the increase in the number of automata.

Using the `Plot graph` dialog (Fig. 5.7, page 43 and Fig. 5.8, page 44) we obtain the graph in Fig. 5.9 (page 44). This graph displays the number of n-state minimised DFAs over n and provides exact values as well as the trend visible in Fig. 5.6. Less apparent from this graph, are the outliers (the single 14, two 12 and single 11 state automata) that were generated.

5.1.5 Evaluation

This section evaluates the GrailKnights package according to the goals laid out in Chapter 3. The last goal is a global goal and so is covered in Section 5.3.

5.1.5.1 Mass-manipulation

In this example, the mass-manipulation was done when the randomly generated NFAs were placed in the database. The minimised DFA and total number of states of that minimised DFA were automatically placed in the correct columns when their corresponding NFA was placed in the table. The result is a table filled with automata that may be data-mined and easily manipulated further. This is more flexible than a file containing a list of the total states generated for each automaton as is usual in Unix scripting. This flexibility may be more of a product of using a database than the mass-manipulation itself, although it should be noted that GrailKnights facilitates the use of the database.

5.1.5.2 Interface

While the dialogs required to run through this example may seem overly much, most of them are optional and after the first one (`Create automata randomly` dialog) only needs one to retrieve the information required.

5.1.5.3 Minimal programming

As has already been mentioned, only the table entry in the metadata XML file is required for this example. A Unix script could be done with the same amount of effort with much more functionality (though less flexibility as noted above). This is more a deficiency of the interface which should include a visual XML editing option so that no file editing is required. Currently the GrailKnights does not conform to this goal.

5.2 Example 2: How many regular languages have succinct NFAs?

5.2.1 Problem Statement

If we extend Example 1 (Section 5.1) to count the number of unique minimised DFAs, the result indicates the number of regular languages that can be represented by succinct NFAs. To find the number of regular languages, we need to eliminate the minimised DFAs that accept the same language by testing the isomorphism of the minimised DFAs with one another.

The expected result is a reduction in the number of minimised DFAs while maintaining the trend established in Example 1, that is to say, a marked decrease in the number of regular languages with the number of states of the minimised DFA nearing $2^n - 1$.

5.2.2 Using GrailKnights

Once again GrailKnights requires the following steps in order to investigate this problem:

- Set up the metadata file,
- calculate isomorphisms,
- potential adjustments and
- visualisation of the results.

GrailKnights requires the user to firstly set up a section of the metadata *XML* file as given in Fig. 5.10. This file provides GrailKnights with database tables and columns. Unlike in Example 1, this is meant purely as storage to indicate which minimised DFAs are isomorphic to other minimised DFAs, that is, accept the same language. The *fm_id* column will contain a number identifying the minimised DFA. The number corresponds to the identifying number in the table *fm*, column *fm_id*. The *isomorphs* column is made up of a collection of these identifying numbers, which correspond to those minimised DFAs that are isomorphic to the one identified in the *fm_id* column.

In order to calculate the isomorphic automata, one uses the `Two Column Query` dialog (Fig. 5.11 through Fig. 5.13, page 47). Firstly one selects the function to run,

```
<table name="isomorph" key="iso_id">
  <col name="iso_id" type="INT" id="notnull"/>
  <col name="fm_id" type="INT" input="true"/>
  <col name="isomorphs" type="CLOB" input="true"/>
</table>
```

Figure 5.10: Example 2: "isomorph" table section of metadata *XML* file

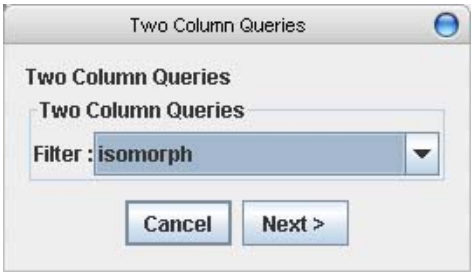

Figure 5.11: Example 2: Two column query dialog, part 1

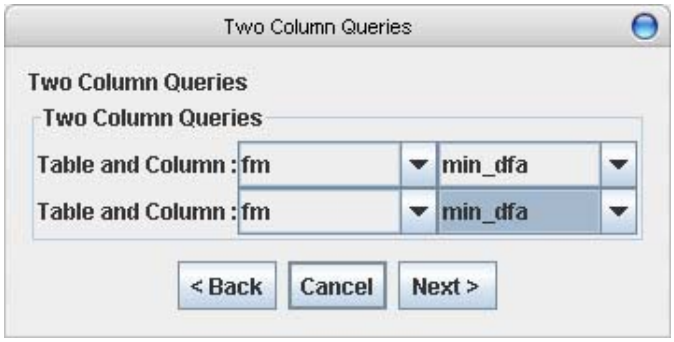

Figure 5.12: Example 2: Two column query dialog, part 2

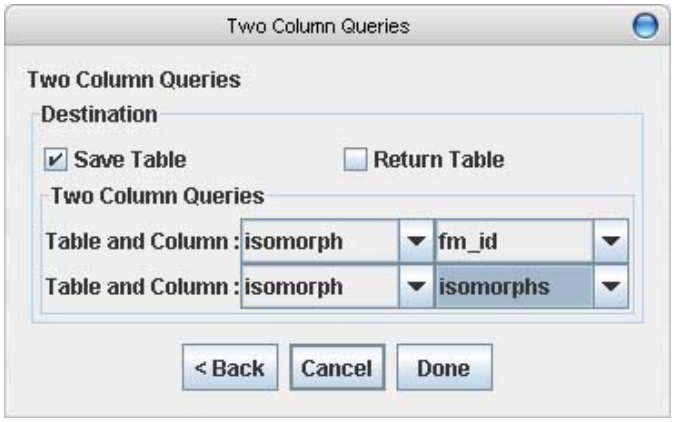

Figure 5.13: Example 2: Two column query dialog, part 3

namely "*isomorph*" and click the `Next` button. Then one selects the two columns that are to be compared. This dialog contains four drop down boxes. The first row indicates the table and column to be compared with the second row's table and column. In this case we select both tables and columns to be the same, namely, "fm" for the tables and "min_dfa" for the columns. Clicking the `Next` button will bring us to the next dialog in which we decide how to return and/or save the returning data. In this case, it is prudent to save our data in our *isomorph* table we defined earlier in the metadata. Making sure that the `Save Table` check box is checked while the `Return Table` check box is unchecked, we need to select into which tables and columns our information will be stored. Once again the tables are on the left and columns are on the right. Both tables, in this case, need to be "*isomorph*" while the top column should be "fm_id" and the bottom column should be "*isomorphs*". Clicking the `Done` button will start the algorithm in the server.

Such a set comparison operation is an $O(n^2)$ operation, that is, polynomial. For large numbers (million or more) computing the isomorphic automata takes a significant amount of time (from days to months). This is one of the advantages of having a client-server model; the client can be closed and leave the server to compute.

Once the server have completed calculating the isomorphic automata, we may view the *isomorph* table using the `Table Query` dialog (Fig. 5.14, page 49). No adjustments are necessary at this point since very little information can be obtained from a simple table view.

5.2.3 GrailKnights Visualisation

We now view the items in a force-directed scatterplot similarly as to what was done in Example 1 above. Additionally we include the isomorphic automata we calculated in this example to the scatterplots output. To do this we use the third part of the scatterplot dialog (Fig. 5.15, page 49) to describe in which table the isomorphic automata are kept as well as specifying which column contains the automatons identifier and which column contains that automatons isomorphs. As one can see in Fig. 5.16 (page 49), the points sort themselves according to the number of total states. The highlighted (orange) points are those that represent unique DFAs while the light greyish blue points are isomorphic to at least one DFA in the highlighted set.

5.2.4 Analysis

Doing the same experiment without the use of the GrailKnights package, that is to say, using only an automaton package such as Grail+, would require the user to first write scripts to do Example 1. Then the user would have to write scripts which identify isomorphism between automata (using Grail+ function *isomorph*) which are kept as a myriad of small text files. These findings would then have to be stored in such a way as to be easily accessible as well as taking into account that calculating isomorphism, as all set comparisons, is a polynomial time operation. This means that the scripts either have to be able to calculate smaller parts of the set

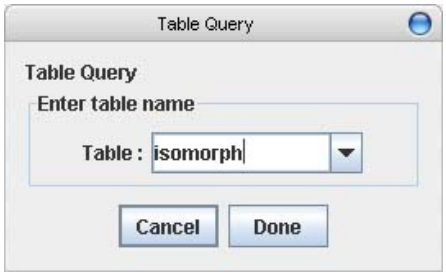

Figure 5.14: Example 2: Table query dialog

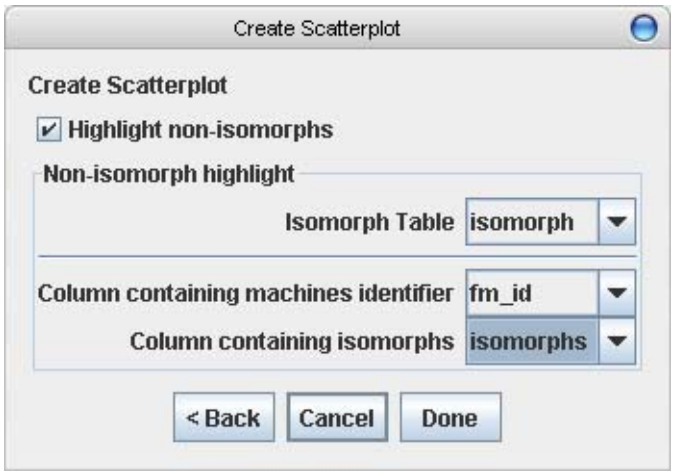

Figure 5.15: Example 2: Create scatterplot dialog, part 3

of automata or have to be robust enough to resume if the calculation were to stop prematurely. The scripts would have to calculate the number of automata with n states, for each n, while adding those that are isomorphic only once. Finally these amounts would have to be graphed using a graphing program. The problems of such a method is similar to those given in Example 1.

As one can see from Fig. 5.16 (page 50), the expected result is apparent: similarly to Example 1, as one nears 15 states the number of automata dramatically decreases. The number of isomorphic automata are most apparent around the low values of n; the most where n equals one and decreasing until they are no longer evident where n equals five. This pattern indicates clearly that for the lower values of n the chance of randomly generating a NFA which accepts a unique language are smaller than those for the larger n.

Similarly to Example 1 we use the `Plot graph` dialog except that we use the third part to exclude the isomorphic automata from the count (Fig. 5.17) by specifing the identities of the automaton and their isomorphic automata. As a result, the graph in Fig. 5.18 (page 51) is obtained. This graph displays the number of n-state minimised

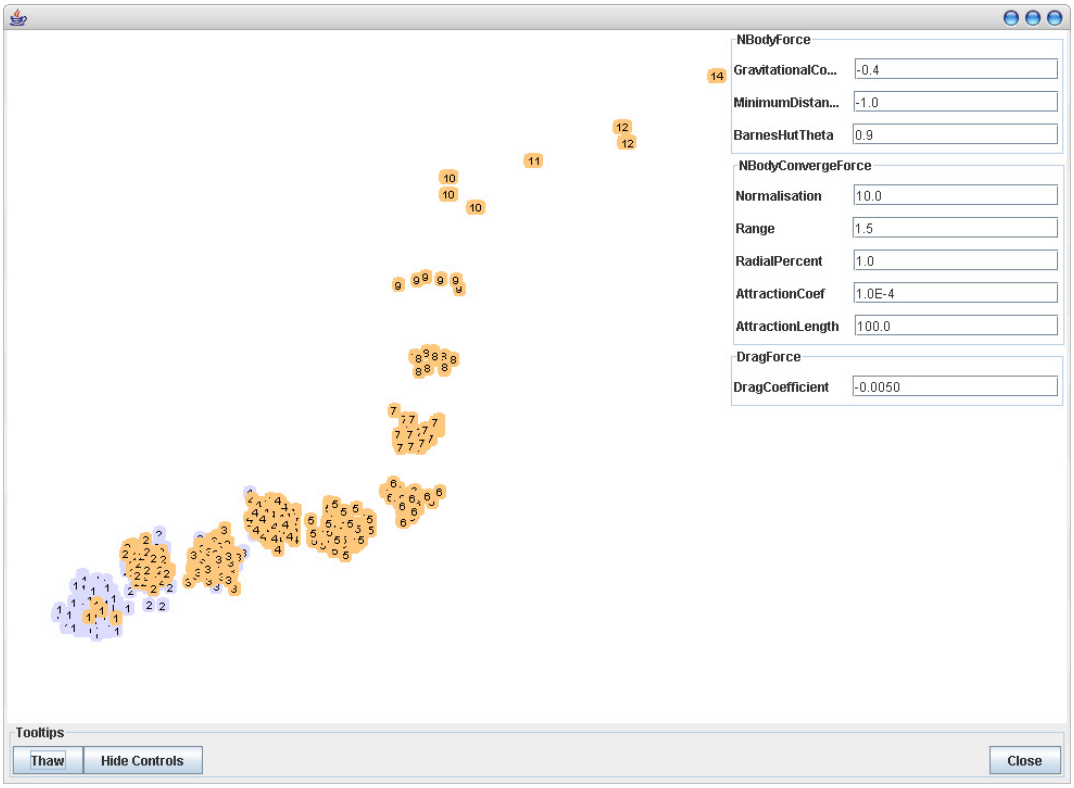

Figure 5.16: Example 2: Scatterplot indicating number of automata with n states

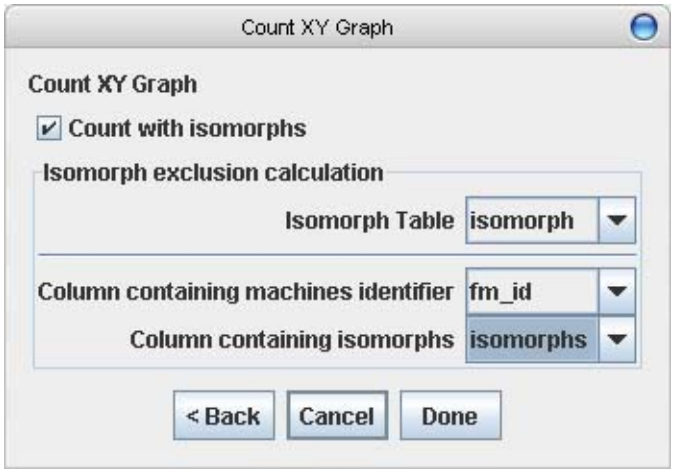

Figure 5.17: Example 2: Plot graph dialog, part 3

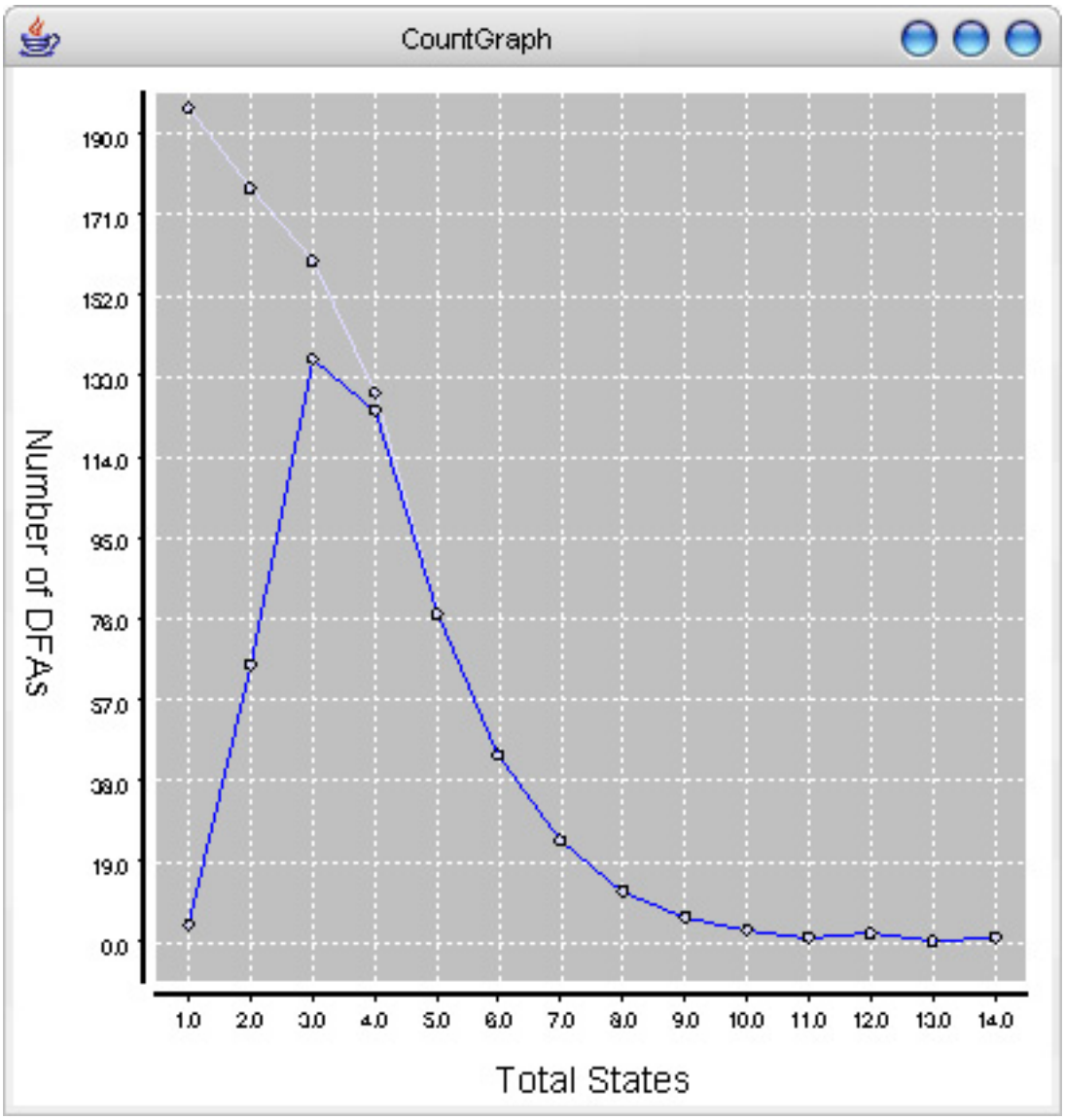

Figure 5.18: Example 2: Graph of number of n-state minimised DFAs over n

DFAs over n and provides exact numbers of automata as well as the trend visible in Fig. 5.16. The number of DFAs that accept unique languages are indicated by the blue line while the graph created in Example 1 is indicated by the grey line.

5.2.5 Evaluation

This section evaluates the GrailKnights package according to the goals laid out in Chapter 3. The last goal is a global goal and so is covered in Section 5.3.

5.2.5.1 Mass-manipulation

In this example, the mass-manipulation was done when calculating the isomorphic automata. Calculating the isomorphic automata is an example of batching where the algorithm used has two inputs. In this example, we compare the set of minimised DFAs to itself. The algorithm is commutative and as such can use all the modifications mentioned in Section 4.5.5. Even with these improvements, its processing time may still be slower than that of its Unix script or C counterpart. The current batching system does make many assumptions of which the user should be aware. These assumptions should be choices made in the user interface as many of these assumptions may be false at times.

5.2.5.2 Interface

In this example, some dialogs show unwanted or unnecessary choices. The user is required to give information that he may not understand such as the 'machines identifier'. A better solution may be for the package to calculate such information for itself.

5.2.5.3 Minimal programming

Only the table entry in the metadata XML file is required for this example (see Section 5.1.5.3).

5.3 Evaluation synopsis

This section looks at GrailKnights as a whole and evaluates the package according to the goals set in Chapter 3 as well as taking into account the evaluations of the examples.

5.3.1 Mass-manipulation

While the GrailKnights package does fulfil the requirement of mass-manipulation, it may not be faster or easier to setup than a Unix script. GrailKnights gains the flexibility of storing data in a database rather than a group of files. The use of a database as well as Unix scripting may be a solution for those interested mass-manipulation but none of the other goals of the GrailKnights package would be met by this solution.

Because the Perceval server uses external executables that contain the algorithms,

the execution of the mass-manipulation is slower than a script using the same executables. This can be overcome by internalising the algorithms, that is, port the algorithms to Java. Though even if the algorithms are internalised, they may run slower than their C counterparts as Java is an interpreting language rather than a faster compiler language such as C.

Currently the ease of use is similar to that of Unix scripting mainly due to the fact that the user is required to edit the metadata XML file in order to build the database tables to their specifications.

5.3.2 Interface

In this section, we see if the GrailKnights package interface conforms to the first principles discussed in Section 2.2. The transparency of the database may not conform to the user's view of the task, however, a database with tables that store large amounts of data in rows and columns is a widely accepted concept. The interface itself is not complex but does allow the user the functionality to run his algorithms *en-masse*. Even so, the masses of dialogs may become tedious to the user and a more interactive interface could be considered in future applications. GrailKnights does promote the learning of the system with its consistent dialogs and a familiar layout. A built in help function which includes the examples in this chapter also encourages learning. Since the database is under the complete control of the user, any mistakes can simply be wiped away by dropping the offending table and running an experiment again. The GrailKnights does not have a method of visualising the database itself, i.e. the tables that are part of the database. This visualisation could form part of the XML editing software which is also lacking at this time. Some, more complex dialogs, such as the `Create Scatterplot` dialog may complicate the users' task needlessly by requiring information that could be obtained programmatically. In GrailKnights it is difficult to distinguish between information and data; since this is an exploration package it is left up to the user to extract information through the use of tools (such as the graph and scatterplot visualisations). Finally the Galahad client is responsive in that the user always has access to the interface. This is due mostly to the fact that any method calls to the server are done on a separate thread. Such a system also allows for parallel processing of client requests.

5.3.3 Minimal programming

The only programming currently required is the editing of the XML metadata and configuration files. This is a good step towards minimalist programming; however a visual XML editor would take this even further. Another drawback is the difficulty of changing the external automata package which could be remedied by creating a generic handler that is modified by editing the metadata and configuration files.

5.3.4 Extensible

The extensibility goal was for the GrailKnights package to be able to change to a user defined database and automaton manipulation package. In Section 4.5.9 we

saw that the database could be changed by editing the configuration file, however, to change the automaton manipulation package it is necessary to create new java code to drive the automaton manipulation package. This is more difficult than originally envisioned and opposes the minimal programming goal for the GrailKnights package. To make this easier a generic class could be written that uses data stored in, for example, the metadata file to handle the automaton manipulation package. Using this method, the automaton manipulation package could be specified in the configuration file and the functioning of the automaton manipulation package could be specified in the metadata file.

5.3.5 Sub-goals

This section looks at GrailKnights as a whole and evaluates the package according to the sub-goals set in Chapter 3. Due to the client-server architecture used, the GrailKnights package may be distributed over two machines: a slower client and a faster server, though this does not prevent the client and server from running on the same machine. The GrailKnights package was designed to fulfil this sub-goal.

The Perceval server encourages the transparency of the database and the automaton packages well but the Galahad client neglects to do so. The GrailKnights package, therefore, only half succeeds at this sub-goal.

Ignoring the licensing issues, the GrailKnights package is easily distributable in that it requires a simple copying of the necessary files. Where the distribution fails is that the configuration file must be edited to reflect the correct directory path to the client and server before the GrailKnights package can be started. This may be remedied by creating an installation program where this may be specified.

The GrailKnights package is written exclusively in Java and as such, will run on any platform that can run the Java virtual machine. However Grail+ is written in C++ and its executable must be changed to one that can run on the current platform. The current version of GrailKnights includes the Windows version of the Grail+ executable. A Linux version of the Grail+ does exist.

5.4 Future work

To take the GrailKnights package from its current prototype to a beta version, would require the following.

- The Galahad client requires an XML editor so that the metadata file and configuration file may be edited using the Galahad client itself instead of having to edit them externally. This editor may form part of a more interactive interface which relies less on a multitude of dialogs and more on object manipulation in the form of trees, etc.
- The current force-directed layouts in the Galahad client are sluggish for a large number of items. This force-directed layout should be optimised or its outputs should be calculated for a set time interval and then displayed.

- The Galahad client currently uses graphs and scatterplots to show relationships between automata. Other visualisations, or alterations on the existing visualisations of automata relationships could be included in a future version.
- While falling outside the scope of this thesis, additional visualisations could be included to view a single automaton.
- The Galahad client uses the Grail+ format for automata exclusively. The Grail+ format is assumed when drawing the visualisation of a single automaton. A generic automaton format converter could be written using formatting data defined in the metadata file.
- Ultimately the Perceval server could include its own automaton manipulation functions which would remove the need for an external automata package. Given the need for an external automata package, the extensibility should be made easier by creating a generic handler that may be modified through the metadata and configuration files.

Chapter 6

Conclusion

The original concept behind the creation of the GrailKnights package was to create an application that will enable researchers to create and manipulate large groups of automata easily and visually discover any relationships between those automata. The application had to maintain each step in the calculation of the various properties of the automata.

In designing the application, various choices were made. The application would use a server-client model. The server would be backed by a database and handle the automata manipulation filters. The client would have a graphical user interface through which the user would manipulate the automata with minimal programming required. The application would be written in an object orientated language so as to take advantage of the modularity inherent in object orientated languages allowing for third parties to extend the application with ease.

The implementation of the application lead to both the client and server being written in Java to take advantage of its ability to function across multiple operating systems as well as its object orientated nature. Cloudscape, a Java-based and freely available database, was used. The automata manipulation functions of Grail+ were used to illustrate the modularity of the application rather than creating new Java code. The graphical user interface uses Java Swing components which are viewable on most operating systems. The graphs are drawn with code based on that of the Prefuse Java application.

Examples were used to illustrate the use of the GrailKnights package. While the GrailKnights package handled these examples amiably, GrailKnights requires the changes mentioned in Future Work (Section 5.4) before being released as a beta version for beta testing by researchers. Further changes may be made depending on the researcher's input.

The GrailKnights package is a proof of concept: while many applications exist to create and manipulate single automata, none, to the author's knowledge, allows for the mass manipulation of automata without having the user resort to some form of coding. The GrailKnights package is an example of such an application.

Appendices

Appendix A

Definitions

The definitions in this section are given by Sipser [37], which provides a good background to automata theory.

A.1 Deterministic finite automaton

Definition 1 A *deterministic finite automaton (DFA)* is denoted by a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$ with

- Q the finite nonempty set of states,
- Σ the finite nonempty input alphabet,
- $\delta : Q \times \Sigma \to Q$ the transition function,
- q_0 the start state, and
- $F \subseteq Q$ the set of final states.

□

The basic structure of a DFA can be seen as a graph with nodes representing states and labelled edges representing transitions. We call this the underlying graph of the DFA. This makes it possible to apply graph theory to investigate the properties of DFAs.

A.2 Regular language

A language which is recognised by a finite automaton is called a regular language. The language which is accepted by finite automaton M is the set of all strings that M accepts. A string u is accepted if the resultant state $\delta(q_0, u)$ is a final state.

A.3 Regular expression

Regular languages can be described by finite automata or by regular expressions. We assume an understanding of regular expressions. For a formal definition, see [37].

A.4 Nondeterministic finite automaton

Nondeterminism is a specialisation of determinism. The key differences between a deterministic finite automaton (DFA) and a nondeterministic finite automaton (NFA) are as follows.

- DFAs have a single start state, while NFAs have a set $Q_0 \subseteq Q$ of start states.
- The DFA transition function operates on an alphabet symbol and state and produces a single state whereas the NFA transition function operates on an alphabet symbol and state and produces a set of states.
- DFA transitions require an alphabet symbol, while NFA transitions can include a transition on the empty symbol. For this reason the alphabet for an NFA Σ_ϵ is defined as $\Sigma \cup \{\epsilon\}$ (see Definition 2 below).

An NFA transition function takes any state q and an alphabet symbol or ϵ and produces a subset of possible next states. According to the binomial theorem (see [37]) there are 2^Q possible different subsets for any given set Q. Here follows a formal definition of a nondeterministic finite automaton.

Definition 2 A *nondeterministic finite automaton (NFA)* is denoted by a 5-tuple $N = (Q, \Sigma, \delta, Q_0, F)$ with

- Q the finite nonempty set of states,
- Σ the finite nonempty input alphabet,
- $\delta : Q \times \Sigma_\epsilon \to 2^Q$ the transition function,
- $Q_0 \subseteq Q$ the set of start states, and
- $F \subseteq Q$ the set of final states.

□

A.5 Isomorphism

Two finite automata are *isomorphic* if there is a renumbering of states such that the finite automata are identical in every way including initial and final states. More formally, finite automata $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$ are *isomorphic* if there is a one-to-one mapping from the state set Q_1 onto Q_2 such that

$$\begin{gathered} \alpha(Q_1) = Q_2 \\ \alpha(\delta_1(q_i, a)) = \delta_2(\alpha(q_i), a) \\ \alpha(F_1) = F_2 \\ \alpha(I_1) = I_2 \end{gathered}$$

for all $q_i \in Q_1$ and $a \in \Sigma$.

Appendix B

Metadata file

B.1 *meta.xml*

This Appendix describes the tags of the metadata XML file and their attributes. Note that if an attribute is not marked as required, it is an optional attribute.

The root of the XML file should contain the references to the schema file (*meta.xsd*) for validation purposes. It is required to contain the `database` attribute which gives the name of the database to be created and used. The `separator` attribute is optional and describes the string used to separate strings that occur in a single attribute, for example: `filter="fmdeterm fmmin"` if using the default separator of a space (`" "`). One could use this to specify other separators for example `separator=","` would be used as follows: `filter="fmdeterm,fmmin"`.

The automata manipulation algorithms are described by the `filter` tags. Each filter has a name, handler, the number of input(s) and output(s) and, finally, the input and output type. The filter tags' attributes are as follows.

- The required `name` attribute defines the name of the filter.
- The `handler` attribute defines which program or library contains the algorithm.
- The `inputNum` and `outputNum` attributes reflect the number of inputs and outputs for the algorithm, respectively.
- The `inputType` and `outputType` describe the filter's input and output types, respectively. The types may be any one of FM (finite machine), RE (regular expression), FL (finite language) or the Java basic class types such as Boolean, Integer, Double and so forth. These types are rigidly defined as most algorithms only accept a single specific type. For algorithms that accept multiple types or with many different options, it may be advisable to create multiple separate filters for a single algorithm and then set the correct options in the automaton handler.

The `filter` data is firstly specified by the user by manully setting up the XML file (along with the rest of the metadata file). The metadata handler then reads in the

data. The Perceval server then uses that data to call the associated algorithm as per specification in the automaton handler. The results are, finally, presented to the user in the Galahad client. Also, for ease of use, the Galahad client, also uses this metadata to build up a list of available algorithms that the user may employ.

The tables in the database are created using the `table` tags. The `table` tags' attributes are as follows.

- The required `name` attribute gives the table's name.
- The `key` attribute(s) define which column(s) in the table are to be used as the primary key(s) for the table.

The only valid tags within the `table` tags are the `col` tag and the `mcol` tag. The `col` tag defines a single column in the table while the `mcol` defines multiple columns at once. The `col` tag contains the following attributes.

- The `id` attribute is a boolean value (usually set to true) denoting that the current column should be generated as an identity column. The corresponding `type` attribute is usually *smallint*, *int* or *bigint*. When a new row is inserted into the table, the value of this column is incremented by 1. This is meant to serve as a unique number describing each of the rows in a table. The increments might not be continuous. Such an identifier column is usually used as the primary key column for the table.
- The `input` attribute is an attribute for when rows are being inserted into the table in conjunction with the tools provided by the GrailKnights package. If a column is marked as an input column, the tool will wait for external input before proceeding. There is usually one input column per table. If no (or more than one) input column is defined, the input and filter attributes are ignored and inserting into such a table requires the user to specify all values.
- The `filter` attribute is an attribute for when rows are being inserted into the table in conjunction with the tools provided by the GrailKnights package. This describes a chain of Grail+ algorithms to be applied to the designated input column to obtain this column's values. To use this attribute, an input column must occur previously to this column in the table.
- The `from` attribute is used in conjuction with the `filter` attribute. This attribute shows which column to use as input for the filters. The column specified must occur previously to this column in the table, to prevent circular prerequisites. If no `filter` attribute is provided, the column is left empty.
- The `from` attribute is used in conjuction with the `filter` attribute. This attribute shows which column to use as input for the filters. The column specified must occur previously in the table, to prevent circular prerequisites. If no `filter` attribute is provided, the column is left empty.
- The required `name` attribute specifies the name of the column.
- The required `type` attribute specifies the SQL type of the column.

- The `gtype` attribute specifies which of the FM (finite machine), RE (regular expression) or FL (finite language) types the column contains. This is used if the `type` attribute type is a non-distinguishing CLOB[1] or BLOB[2] type which is most often used to store large streams of data.

The multiple column tag (`mcol`) is only used for those filters that have one input and more than one output. The text between tags lists the multiple columns' names separated by the separator defined in the beginning of the file. The multiple column tag contains the following attributes.

- The `number` attribute gives the number of columns that this multiple column tag represents as well as the number of names that should be given in the text between tags.
- The `type` attribute describes the SQL type of all the columns.
- The `filter` attribute is the filter used which has a number of outputs equal to the value given in the `number` attribute.
- The `from` attribute gives the name of the column which the filter uses as an input, similarly to the `filter` attribute given in the `col` tag.
- The `filterIgnore` attribute is a list of numbers which are used to ignore those numbered outputs (starting at 1). This is usually used when the number of outputs of an algorithm is larger than the number of columns wanted. In the example (Fig. B.1, page 63), the Grail+ algorithm, `fmstats`, has five outputs, but only the first three are wanted, so the fourth and fifth outputs of the filter are ignored.

[1] An SQL Character Large Object
[2] An SQL Binary Large Object

```
<metaRoot
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation='meta.xsd'
database="MyDB"
separator=" "
>
  <filters>
    <filter name="isomorph" inputNum="2" inputType="FM"
      handler="grail" outputNum="1" outputType="Boolean"/>
    <filter name="fmmin" inputNum="1" inputType="FM"
      handler="grail" outputNum="1" outputType="FM"/>
    <filter name="fmdet" inputNum="1" inputType="FM"
      handler="grail" outputNum="1" outputType="FM"/>
    <filter name="fmstats" inputNum="1" inputType="FM"
      handler="grail" outputNum="5" outputType="Integer"/>
  </filters>

  <table name="fm" key="fm_id">
    <col name="fm_id" type="INT" id="notnull"/>
    <col name="fm" type="CLOB" input="true" gtype="fm"/>
    <col name="min_dfa" type="CLOB" filter="fmdeterm fmmin"
     gtype="fm"/>
    <mcol number="3" type="INT" filter="fmstats"
     from="min_dfa" filterIgnore="4 5">
      states_start states_final states_total
    </mcol>
  </table>

  <table name="isomorph" key="iso_id">
    <col name="iso_id" type="INT" id="notnull"/>
    <col name="fm_id" type="INT" input="true"/>
    <col name="isomorphs" type="CLOB" input="true"/>
  </table>

</metaRoot>
```

Figure B.1: Metadata example: `meta.xml`

Appendix C

Configuration file

C.1 *conf.xml*

This Appendix describes the tags of the configuration XML file and their attributes. It should be noted that those attributes that are not marked as required are optional.

The root (`configRoot` tag) of the XML file should contain a reference to the schema file (*conf.xsd*) for validation purposes. The tags in the configuration file each have a required `name` and a required `value` attribute that represent the name and value of the tag, respectively. The root tag does not have a `name` or `value` attribute. The configuration file contains the following tags.

- The `conf` tags are the first tags to be defined in the XML file. If a `value` attribute occurring later in the configuration file contains a term consisting of a string between curly brackets (`{}`), then the entire term is replaced with the `value` attribute of the `conf` tag. For example, if `<conf name="X" value="B"/>` is specified initially and later followed by `<str name="str1" value="A{X}C"/>`, the value of the property `str1` will become `"ABC"`. Global properties are any tags (except for `conf` tags) specified before the class tags.
- The `str`, `int` and `bool` tags describe strings, integers and booleans, respectively; the `value` is of that type.
- The `handler` tag is used to describe which Java class is used for which algorithm application. The `name` attribute should match the `filter` attribute in the metadata file.
- The `class` tag provides class specific information of the GrailKnights package. Each class has a `debug` attribute which defaults to false if not specified. A `class` tag does not have a `value` attribute and the `name` is renamed to `className` for clarity. The `className` tag does not need to specify the Java namespace due to the uniqueness of the class names being used. The <name, value> pairs given under a specific `class` tag are considered hierarchical, that is, if a name is not found under a class list of names, then the global names are used.

There is a single configuration file for both the Perceval server and Galahad client. An example of the configuration file is given in Fig. C.1 (page 65) and Fig. C.2 (page 66).

```
<configRoot
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation='config.xsd'
>
<conf name="baseDir" value="c:\test\Perceval\"/>
<conf name="dbDir" value="c:\test\test\db\"/>
<conf name="galahadDir" value="c:\test\Galahad\"/>
<str name="rmipolicy" value="{baseDir}conf\perc.policy"/>
<str name="rmihost" value="localhost"/>
<str name="rmiInstance" value="PercevalInstance"/>
<int name="rmiport" value="1099"/>
<bool name="localRegistry" value="true"/>
<handler name="grail" value="perceval.handler.automata.GrailHandler"/>
<class className="Perceval" debug="true">
  <str name="DBHandler" value="perceval.handler.database.CloudNine"/>
</class>
<class className="RandomFiniteMachineFactory" debug="true">
  <str name="density" value="25"/>
  <str name="final_density" value="50"/>
  <str name="states" value="5"/>
  <str name="symbol_count" value="2"/>
</class>
<class className="HandlerManager" debug="true">
</class>
<class className="GrailHandler" debug="false">
  <str name="tempDir" value="{baseDir}"/>
  <str name="grailDir" value="{baseDir}grail\"/>
  <int name="sleep_msc" value="1000"/>
  <int name="timeout" value="60"/>
</class>
```

Figure C.1: Configuration file example: `conf.xml` part 1

```
<class className="CloudNine" debug="true">
  <str name="framework" value="embedded"/>
  <str name="driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
  <str name="protocol" value="jdbc:derby:"/>
  <str name="user" value="user1"/>
  <str name="password" value="user1"/>
  <str name="dbhome" value="{dbDir}"/>
</class>
<class className="Meta" debug="false">
  <str name="metafile" value="{baseDir}conf\meta.xml"/>
</class>
<class className="InputStreamer" debug="true">
  <str name="seperator" value=";"/>
  <str name="newline" value=" "/>
  <str name="table" value="fm"/>
</class>
<class className="Logger" debug="true">
  <str name="logfile" value="{baseDir}Perceval.log"/>
</class>
<class className="StreamTracker" debug="false">
</class>
<class className="Config" debug="false">
</class>
<class className="MetaHandler" debug="false">
  <str name="seperator" value=" "/>
</class>
<class className="QueryHistory" debug="false">
  <str name="query_history" value="{galahadDir}conf/query.xml"/>
</class>
</configRoot>
```

Figure C.2: Configuration file example: `conf.xml` part 2

Bibliography

[1] *Quest for the Holy Grail.*
Available at: `http://www.lib.rochester.edu/camelot/mainmenu.htm`

[2] *tk-widget.*
Available at: `http://www.tk.org and http://www.scriptics.com/software/tcltk/`

[3] *Understanding Metadata*, 2004. National Information Standards Organization.
Available at: `www.niso.org/standards/resources/UnderstandingMetadata.pdf`

[4] Avgeriou, P.: *Architectural Patterns Revisited A Pattern Language*, 2005. Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, pp. 1–39.

[5] Bederson, B. and Shneiderman, B.: *The Craft of Information Visualization.* Morgan Kaufmann Publishers, 2003.

[6] Booch, G.: *Object Orientated Analysis and Design, with Applications.* 2nd edn. John Wiley and Sons. Inc., 1997.

[7] Budd, T.A.: *An Introduction to Object-Orientated Programming.* 2nd edn. Addison-Wesley, 1997.

[8] Carrol, J.M.: *HCI Models, Theories, and Frameworks Toward a Multidisciplinary Science.* Morgan Kaufmann Publishers, 2003.

[9] Champarnaud, J.M. and Hansel, G.: *AUTOMATE, A Computing Package for Automata and Finite Semigroups.* Journal of Symbolic Computation, vol. 12, no. 2, pp. 197–220, 1991. ISSN 0747-7171.

[10] Di Battista, G., Eades, P., Tamassia, R. and Tollis, I.G.: *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, Upper Saddle River, NJ 07458, 1999.

[11] Dubois, J.P.: *Java RMI*, 2007. Bern University of Applied Sciences.
Available at: `https://staff.hti.bfh.ch/doj1/rmi_Jini/doc/rmi/RMI.pdf`

[12] Elliott, J., Eckstein, R., Loy, M., Wood, D. and Cole, B.: *Java Swing.* 2nd edn. O'Reilly Media, Inc., 2002.

[13] Frishert, M.: *FIRE Works & FIRE Station: a Finite Automata & Regular Expression Playground*, 2005. Eindhoven University of Technology.
Available at: `http://www.fastar.org/publications/MScMichiel.pdf`

[14] Goodrich, M.T. and Tamassia, R.: *Data Structures and Algorithms in Java.* 2nd edn. Benjamin/Cummings, 1994.

[15] Gosling, J. and McGilton, H.: *The Java Language Environment. A White Paper. Sun Microsystems*, 1996.
Available at: `http://java.sun.com/docs/white/langenv/`

[16] Harold, E.R.: *Processing XML with Java(TM): A Guide to SAX, DOM, JDOM, JAXP, and TrAX.* 1st edn. Addison-Wesley Professional, 2002.

[17] Harold, E.R. and Means, W.S.: *XML in a Nutshell.* 3rd edn. O'Reilly Media, Inc., 2004.

[18] Healey, C.G., Booth, K.S. and Enns, J.: *Visualizing Real-Time Multivariate Data Using Preattentive Processing.* ACM Transactions on Modeling and Computor Simulation, vol. 5, no. 3, pp. 190–221, 1995. ISSN 1049-3301.
Available at: `http://doi.acm.org/10.1145/217853.217855`

[19] Heer, J.: *Prefuse*, 2005. Prefuse@jheer.org.
Available at: `http://prefuse.sourceforge.net`

[20] Helander, M.: *Handbook of Human-Computer Interaction.* Elsevier Science Ltd., North-Holland, 1988.

[21] IBM: *Cloudscape 10*, 2005.
Available at: `http://www-306.ibm.com/software/data/cloudscape/`

[22] Johnson, J.: *GUI Bloopers. Don'ts and Do's for Software Developers and Web Designers.* Morgan Kaufmann Publishers, 2000.

[23] Kanthak, S. and Ney, H.: *FSA: An Efficient and Flexible C++ Toolkit for Finite State Automata Using On-Demand Computation.* ACL '04: Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics, p. 510, 2004.
Available at: `http://dx.doi.org/10.3115/1218955.1219020`

[24] Klinger, A.: *Human-Machine Interactive Systems.* Plenum Press, 1991.

[25] Krieg-Brückner, B.: *uDraw(Graph) previously known as daVinci.* University Bremen.
Available at: `http://www.informatik.uni-bremen.de/uDrawGraph/en/download/`

[26] Lansdale, M.W. and Ormerod, T.C.: *Understanding Interfaces.* Academic Press, 1995.

[27] Lombardy, S., Regis-Gianas, Y. and Sakarovitch, J.: *Introducing Vaucanson.* Theoretical Computer Science, vol. 328, pp. 77–96, 2004.

[28] Matz, O., Miller, A., Potthoff, A., Thomas, W. and Valkema, E.: *Report on the Program AMoRE.* Institut fur Informatik und Praktishe Mathematik der Christian Albrechts Universitat zu Kiel, 1995. Technical Report 9507.
Available at: `http://citeseer.ist.psu.edu/193723.html`

[29] Mehlhorn, K.: *The LEDA User Manual Version 4.0.* Max-Planck-Institut für Informatik, Saarbrücken, Germany.
Available at: `http://sun.ac.za/mecheng/MD314`

[30] Meyer, B.: *Object-Orientated Software Construction.* 2nd edn. Addison-Wesley, 1998.

[31] Raymond, D. and Wood, D.: *The Grail Papers: Version 2.0.* University of Waterloo, Canada, 1996.
Available at: `http://softbase.uwaterloo.ca/drraymon/papers/grail20.ps`

[32] Reichert, R., Nievergelt, J. and Hartmann, W.: *Programmieren mit Kara. Ein spielerischer Zugang zur Informatik.* 2nd edn. Springer, 2004.

[33] Renaud, P.E.: *Introduction to Client/Server Systems.* Wiley, 1996.

[34] Rodger, S.H. and Finley, T.W.: *JFLAP: An Interactive Formal Languages and Automata Package.* 1st edn. Jones & Bartlett, 2006.

[35] Sander, G. and Lemke, I.: *Visualization of Compiler Graphs.* 1993. Design Report. Available at: `http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html`

[36] Schneiderman, B. and Plaisant, C.: *Designing the User Interface. Strategies for Effective Human-computer Interaction.* 4th edn. Pearson Addison-Wesley, 2005.

[37] Sipser, M.: *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, 1997.

[38] Tanin, E., Shneiderman, B. and Xie, H.: *Browsing Large Online Data Tables Using Generalized Query Previews.* Information Systems, vol. 32, no. 3, pp. 402–423, 2007. ISSN 0306-4379.
Available at: `http://dx.doi.org/10.1016/j.is.2005.12.006`

[39] Tscherter, V.: *Exorciser: Automatic Generation and Interactive Grading of Structured Exercises in the Theory of Computation*, 2004. Master's thesis, Swiss Federal Institute of Technology, Zurich, Switzerland.
Available at: `http://e-collection.ethbib.ethz.ch/`

[40] Van der Veer, G.C. and Mulder, G.: *Human-Computer Interaction: Psychonomic Aspects.* Springer-Verlag, Berlin, 1988.

[41] Van Zijl, L., Harper, J. and Olivier, F.: *The MERLin Environment applied to *-NFAs*, 2001. Proceedings of the CIAA 2000, July 2000, London, Ontario, Canada. Lecture Notes in Computer Science Vol. 2081.

[42] Watson, B.W.: *An Introduction to the FIRE Engine: A C++ Toolkit for FInite Automata and Regular Expressions.* 1994.
Available at: `http://citeseer.ist.psu.edu/67829.html`

[43] Watson, B.W.: *Taxonomies and Toolkits of Regular Language Algorithms.* 1995. Ph.D. thesis, Eindhoven University of Technology.
Available at: `http://www.fastar.org/publications/PhD_Watson.pdf`