The Ballarat Incremental Knowledge Engine

Richard Dazeley[1], Philip Warner[2], Scott Johnson[3] and Peter Vamplew[1]

Graduate School of Information Technology and Mathematical Sciences, University of Ballarat, University Drive, Mount Helen, Victoria 3353, Australia
[1]{r.dazeley, p.vamplew}@ballarat.edu.au, [2] pjw@rhyme.com.au, [3]scjohnno@gmail.com.

Abstract. Ripple Down Rules (RDR) is a maturing collection of methodologies for the incremental development and maintenance of medium to large rule-based knowledge systems. While earlier knowledge based systems relied on extensive modeling and knowledge engineering, RDR instead takes a simple no-model approach that merges the development and maintenance stages. Over the last twenty years RDR has been significantly expanded and applied in numerous domains. Until now researchers have generally implemented their own version of the methodologies, while commercial implementations are not made available. This has resulted in much duplicated code and the advantages of RDR not being available to a wider audience. The aim of this project is to develop a comprehensive and extensible platform that supports current and future RDR technologies, thereby allowing researchers and developers access to the power and versatility of RDR. This paper is a report on the current status of the project and marks the first release of the software.

Keywords: Ripple Down Rules, Toolkit, Knowledge Based System, Machine Learning

1 Introduction

Knowledge based systems (KBS) have become a common inclusion in many information processing systems. While early Expert Systems (ES) tended to be monolithic stand alone entities, the modern KBS tends to sit inside a larger system providing specialized functions for particular processes. This has allowed for an increase in the use of KBS technologies; however many of these are using traditional ES approaches that are not easily extended or maintained. These systems therefore have a limited life, which intern limits the life span of the system it is embedded.

The Ripple Down Rules (RDR) [1] family of methodologies have been widely recognized as a powerful production rule approach that addresses these issues. Research and development in RDR has been pursued for an extended period of time resulting in many refinements and commercial systems. However, researchers and developers have always developed their own implementations, which meet their direct need [2]. While some researchers have released their implementations for others to use, these are generally not easily extendable or applicable in other applications. Commercial implementations, as expected, have always been controlled by companies under strict licensing and are unavailable [2].

This paper introduces an engine, referred to as the Ballarat Incremental Knowledge Engine (BIKE)[1] that is a comprehensive and extendable platform specifically designed for the RDR family of methodologies. The intention was that the engine will serve two primary purposes. The first was that the system should facilitate future research in RDR by being extendable and versatile. The second was for the system to provide a platform for developers to incorporate incrementally maintainable KBS solutions by including database integration services.

To accomplish these goals the engine was designed using a plugin architecture that allows any aspect of the system's behaviour to be overridden. For instance, changes to knowledge structure, the inferencing process or learning methodology can all be easily extended and modified. The following section will provide an overview of the various RDR approaches, followed by a brief discussion on implementations of RDR. Section 4 will discuss the first release of BIKE, focusing on the services it provides and aspects that allow it to be extended. Finally, we will discuss future extensions to the engine and discuss how researchers and developers can contribute to this project.

2 Overview of Ripple Down Rules (RDR)

Ripple Down Rules (RDR)[2] was first suggested by [1] as an approach to resolving the maintenance problem in knowledge based systems. It was argued that experts do not explain how a conclusion is reached; rather, that they justify their conclusion within a particular context [1, 2]. RDR was designed to capture this contextual information by storing the knowledge in an exception structure, where it was assumed that the context was the sequence of rules that were evaluated in reaching a conclusion [1, 3, 4]. This situation cognition view of knowledge [6, 7] resulted in the ability to capture expert's knowledge incrementally. Furthermore, the design of the methodology allows an expert system under development to validate knowledge without the need of a knowledge engineer or expensive testing procedures [4].

RDR uses a binary exception tree, where each node contains a rule, a conclusion, a cornerstone case and two branches: labeled as the 'true' (or 'exception') branch and a 'false' branch. During inferencing if a rule at the current node is found to be true then the true branch is followed and vice versa if the rule is false [1]. This process continues until a node is reached with no child down the appropriate branch. The conclusion returned is the one associated with the last successful rule.

For example (adapted from [7]), a case with the attributes {a, b, c, g, h} is presented to the RDR KB shown in Fig 1. In this tree it can be seen that: rules 1 and 3 have both true and false branches leading to further rules; rule 2 only has a false path; and, rule 6 (and the root node) only has a true path. When the case is presented it ripples down the tree using the path {0 – 1 – 3 – 6} where, because there is no attribute 'f' and no false branch, the inferencing process completes. The conclusion returned is 1 from rule 1, due to this being the last rule satisfied.

Learning in RDR is performed using a failure-driven approach [2] where the expert corrects a conclusion with which they disagree. After identifying a misclassification

[1] Documentation, software and source available at http://bike.ballarat.edu.au/.
[2] RDR is sometimes referred to as Single Classification RDR (SCRDR).

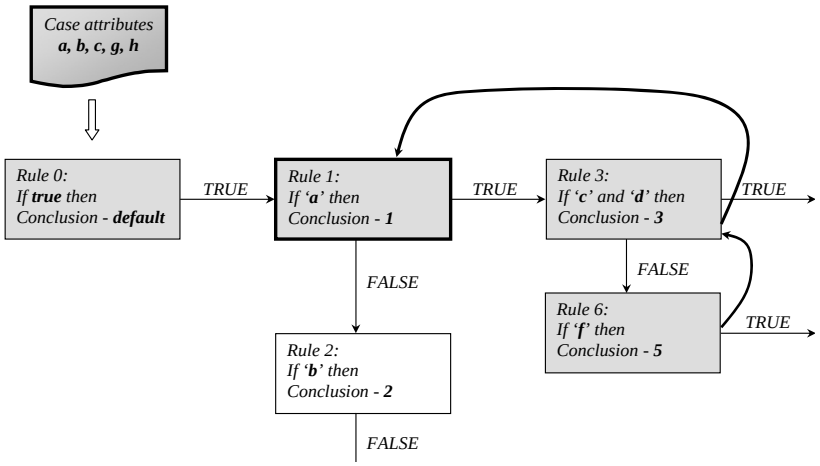

Fig 1. Example of the RDR binary tree structure and inferencing process [7].

the expert simply provides the correct conclusion and a justification for why the original response was wrong. The justification is determined by the system first comparing the current case with the cornerstone case held at the node being corrected. A list of differences between these two cases is generated from which the expert selects one or more. The attributes selected by the expert are then used to justify the new rule. The new node includes a rule, the conclusion given by the expert and the case just processed as the new cornerstone case [1].

For example, Fig 2 illustrates how a new rule is created and added when the expert has decided the conclusion of class 1 is incorrect. Firstly, the cornerstone case from rule 6 is loaded and a difference list is extracted. The expert then selects the relevant differences that best distinguish between the documents, for instance 'h' and '!i' to form the new rule. A new node is then attached as child of rule 6 on the false branch containing the rule, the correct class given by the expert and our current case. The current case will become the cornerstone case for this new rule.

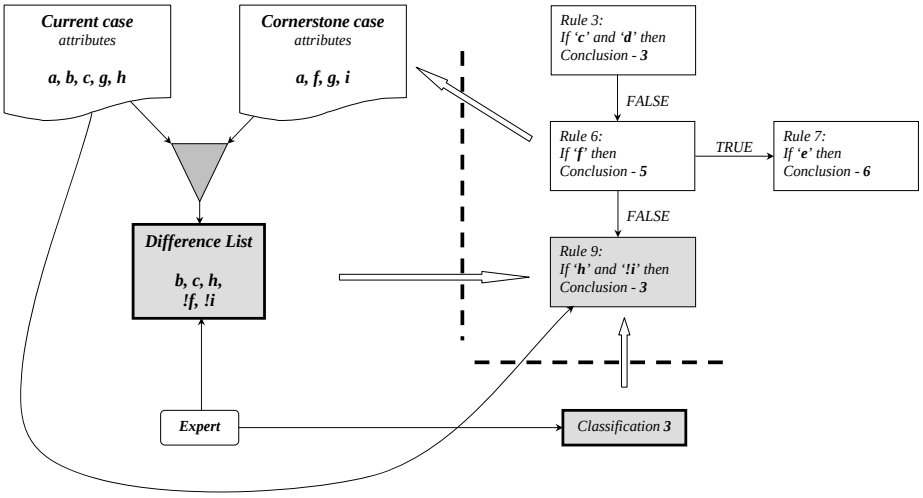

Fig 2. Example of creating and incorporating new knowledge in RDR.

Since the creation of the original RDR there have been numerous extensions, modifications or new methodologies using the same philosophical basis. For instance, one popular adaption is Multiple Classification Ripple Down Rules (MCRDR) [8]. MCRDR is designed to be capable of producing multiple conclusions for each case, by using an n-ary tree where each child branch represents an exception to the parent [9]. Other approaches have been developed such as Nested RDR (NRDR) [10], Time Course RDR (TCRDR) [11], WISE [9] Dynamic RDR (DRDR) [12], Ripple down rule-Oriented Conceptual Hierarchies (ROCH) [13], MCRDR/FCA [14], Collaborative RDR (CRDR) [15] and Rated MCRDR [16, 17] to name just a few. The above approaches have been applied in a number of applications such as Labwizard by Pacific Knowledge Systems [18], KMAgent [2, 19], Knowledge Management Assistant (EMMA) [20] or embedded in other systems such as in a conversational agent [21] and planning [22]. RDR approaches have also been applied in machine learning through such techniques as InductRDR [23] and Cut95 [24]. For a more complete discussion readers are directed to the recent survey by [2].

3 Current implementations

RDR methodologies have been implemented by several researchers and organizations over an extended period of time. The majority of researchers have developed their implementation from scratch to be used in their particular project [2]. Some of these researchers at various times have made these publicly available, such as Suryanto [2]. One of the most professionally developed RDR engines made publicly available was released by Associate Professor Byeong Kang and the MCRDR research group from the University of Tasmania in 2004[3]. This engine is a solid multiplatform implementation of MCRDR in Java and C. Unfortunately, these implementations are not easily extensible to new versions of the algorithm. Furthermore, they have not been updated or maintained since their initial release.

There have also been numerous commercial developments of the RDR engine. For instance the Pathology Expert Interpretative Reporting System (PEIRS) [25], LabWizard developed by Pacific Knowledge Systems in 1996 [26], the Sonetto system developed by the Ivis Group[4], and the Yawl group that used RDR in their workflow management system [3]. The RDR engines at the core of these developments however have not been released for researchers or developers to utilize.

4 Ballarat Incremental Knowledge Engine

Work on BIKE started in 2008 to develop a simple RDR implementation to be used in current research and consultancies being performed by the University of Ballarat (UB) research staff. A partial implementation was first deployed in a decision support project for the Victorian Department of Justice in early 2009. Later in 2009 the

[3] http://www.appcomp.utas.edu.au/users/bhkang/
[4] http://www.ivisgroup.com/

project received funding through a Research Infrastructure Block Grant (RIBG) from UB to more extensively develop the system into a general engine. In 2010 the decision was made to release this engine publicly to allow researchers and students free access. The following sections will provide an extensive overview of the engines design and capabilities. It is not the aim of this paper to provide details of how to use or develop with the engine; such details will be made available on the engine's website http://bike.ballarat.edu.au/. This site includes a wiki based repository for researchers and developers to provide details of their work and articles published.

4.1 Overall Design

BIKE was designed from the ground up to be extendable. The intention was to be able to change any and all aspects of the system's operations through the use of plugins. The final design is sufficiently versatile that it could be used to implement many non-RDR based methodologies as well. Fig 3 provides a detailed overview of the system's architecture and identifies the central components of BIKE that have been implemented directly or via plugins. The system contains four primary components: the engine core, knowledge base, stream processing unit and a virtual expert. All of these components have been extended in a number of plugins. The plugins provided in the first release include a full implementation of both RDR and MCRDR, as well as extensions for both human and simulated experts. Other plugins have been included, such as the stream processor plugins and knowledge base plugins, as well as a simple user interface, which provide valuable support functionality.

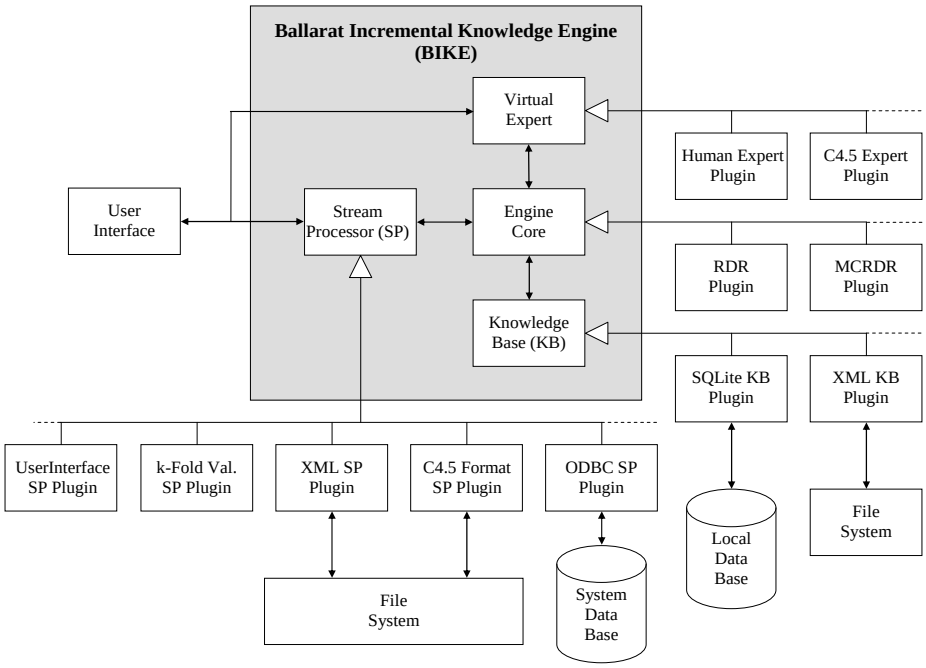

Fig 3. Diagram illustrating BIKEs overall architecture.

4.2 Engine Core

The engine core component provides the majority of the functionality for the system. The primary services provided by this component are the knowledge representation, plugin manager, expression manager, and the inference engine. It also provides a number of the general classes required for processing such as rules, cases, attributes, values, frames and results. The basic operation of this component is to receive a case from the stream processor to be processed by the nominated inference engine along with a knowledge base. The inference engine will process the case using its inference rules to guide its path through the knowledge representation and produce a result. This result is then passed back to a stream processor for post processing. Each of these elements will be described in more detail in the following subsections.

Knowledge Representation

Approaches to knowledge representation in RDR can be very diverse making the development of a generic system difficult. To get around this the core engine only implements a basic framework for how knowledge is represented. Plugins are then used to implement specific features of a particular methodology. The basic structure is based around an n-ary tree. Each `Node` in the tree can have any number of rules and any number of child nodes. Using this structure:

- In RDR the first child node can be regarded as an exception (true) branch and the rest of the children are the false nodes of the node before it, therefore, node *n+1* is the false child of node *n*. During inference RDR simply tests each child until one is found to be true, which is then followed. While this representation does not match the usual way it is conceived it does in fact match the original description of RDR [1].
- In MCRDR each child is simply regarded as an exception to the parent. Traversal is identical to RDR except instead of stopping when you find a child that evaluates to true you continue until all children have been tested, following each child that is true.
- A traditional KBS could also be represented by having a list of rules as a child of a place holding root node. It is then up to the inference engine to determine how the nodes will be traversed.

A planned addition for a future release is to also prove a `GraphNode` with a list of input and output arcs. Such a structure could be used to represent any graph like structure effectively allowing the addition of knowledge representations such as Collaborative RDR (CRDR) [15] or even concept graphs and semantic networks.

Inference Engine

The inference engine is the primary processing unit in the engine. It takes a case provided by an input stream and a knowledge base using the above representation scheme and is responsible for generating a result and sending it to an output stream. In BIKE the Inference Engine is represented with the `IAlgorithm` interface and by default does nothing. Plugins are responsible for implementing the inference engine functionality. However, while each plugin must implement their own version of the algorithm an array of services are provided for them to utilize, as discussed in the following subsections.

Attributes and Values

An `Attribute` is a simple class representing a name-value pair. A collection of attributes makes up a case and they are used in rules and difference lists. The attribute's name is a simply an identifier. There are a number of types of values that extend the interface `IValue`, represented in BIKE, which can also be defined within a plugin. The value types provided currently are:

- `BooleanValue`: A simple Boolean type.
- `IntegerValue`: Represents a value of type integer with up to 64 bits.
- `RealValue`: A value with double floating point precision
- `StringValue`: A value of type string using the Unicode character set.
- `ListValue`: A value that contains a list of other values.
- `Frame`: A Frame is a value that is based on a structured type in object oriented programming languages and will be discussed in the following subsection.

Each of these value types has a number of operations available. For instance `IntegerValue` and `RealValue` types have access to an array of operators: +, -, *, /, -, ^, comparison operators, min, max, sin, cos, tan, asin, acos, atan, atan2, round, DegToRad and RadToDeg. In these operations `IntegerValue`s are promoted to `RealValue`s where required. Likewise strings have comparison operators and concatenation, while lists have a collection of untyped meta-functions, such as ForAll(x in[...], <condition>), ForEach(...), ForAny(...), Count(...), and many more. Additionally, new `IValues` and associated operators can be easily created in a plugin which can be registered with the engine and used. For instance, a plugin could create a `ComplexValue` class and a `ComplexTypeDefinition` class for its operations. This new `IValue`, once registered, can be used with any algorithm.

Frames

One interesting type of `IValue` that has been included is the `Frame`. The frame value type allows the inclusion of structured types. A frame allows a value to be made up of a number of named `FrameSlot`s, which in turn contain an `IValue`. Generally structured facts such as Frames are not used in RDR because it introduces issues in how to generate difference lists and how to build rules from such values. One exception to this is the work by [13] on ROCH which uses conceptual hierarchies. While none of the current plugins for BIKE use the frame facility, it was included to facilitate the potential development of systems like ROCH.

Cases

A `Case` is a class on which all inference engines operate. A case is essentially a collection of attributes that represent the state of the world for a given situation. In different situations a case may be used in various ways. For instance they may be loaded from a file either singly or in a batch process. Alternatively, the case can be created by loading data from a system database or by being entered by a user. The method the case is created however is unimportant to the engine core which simply takes a case via a stream processor (4.5) and applies the inference engine.

Rules and Expressions

A `Rule` is a `Persistable` class that maintains an association between an `Expression`, a `Conclusion` and, if required, a list of `CornerstoneCase`s. While RDR only requires a single cornerstone case at each rule, MCRDR often may store multiple cornerstone cases at a node. The `Conclusion` class is also `Persistable` and contains an `IValue` representing the conclusion to be returned.

The `Expression` class is responsible for performing the evaluation of the rule. It represents an expression tree that contains any number of sub expressions. This structure exceeds the basic requirements of an RDR rule but allows for potential systems that require more advanced expressions. There are numerous types of expressions provided, which can be combined in any way required. The entire expression processing component of the engine is handled by the `ExpressionManager`. The `ExpressionManager` manages operator and function implementations, as well as the parsing of expressions. It stores the various types used by the evaluation process, which are registered with the `ExpressionManager` allowing any number of new types to be added. For instance, fuzzy logic based operations can easily be created and registered with the `ExpressionManager` allowing fuzzy expression resolution.

Results

A `Result` class is returned from each inference and contains details about what occurred. Primarily it contains two pieces of information: The path (sequence of rules) followed during inferencing, as well as the `Conclusion` found. In RDR each inference returns a `Result` with only one path and one `Conclusion`, while MCRDR will return a list of `Result`s with corresponding `Conclusions`.

4.4 Knowledge Base

The Knowledge Base (KB) represents the storage layer for the engine. This component is responsible for ensuring all `persistable` objects are maintained in permanent storage. This component manages and maintains the knowledge base both on the permanent storage as well as in active memory. This design is particularly advantages to the development of large knowledge bases. The storage process is handled by the `IStorageManager` using interchangeable `StorageBackend` objects to store data and manage local in-memory caching. The storage manager knows about all `Persistable` objects through a smart `StoragePointer` that all objects created in the system use. These smart pointers also manage garbage collection allowing components not being used to be released dynamically.

The `IStorageBackend` interface has been extended in two separate plugins. The first of these provides the facility to write the KB in an XML file. The second stores the KB in a SQLite database. Also included is a general SQL backend that allows for a general ODBC backend layer, although the ODBC backend itself is not included in this release. An ODBC backend has the advantage that the knowledge base could be integrated with existing systems. Once again it is a relatively simple process to provide a plugin to control where and how a KB is to be stored.

4.5 Stream Processing

The Stream Processing (SP) component of BIKE provides and manages all processing operations and is key to providing BIKE's versatility. SPs all receive a stream of cases from some source, manipulate them in some fashion and then forward them on. The input and output of an SP can be another SP; therefore SPs can be piped together in numerous arrangements. Furthermore, everything in the BIKE processing life cycle is an SP. For instance the `Classifier` SP manages the entire process of operating the engine core. The `Classifier` SP accepts cases from some source (another SP) and passes it to the inference engine algorithm for classification. This design of attachable SP components can be a very powerful facility. For instance, it could be used for a propose and refine task by piping a series of `Classifier` objects in a series. There are three types of SPs available in BIKE: input, output and filters. BIKE requires at least one input, one output and ideally should have at least one filter but can have more. Fig 4 illustrates an overview of how the BIKE life cycle operates.

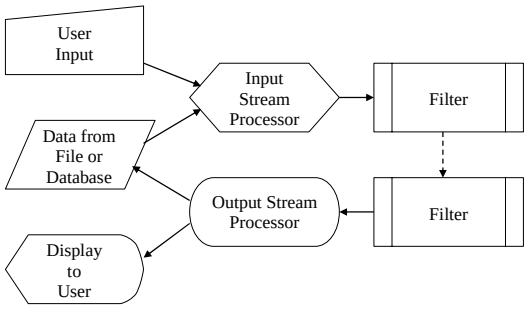

Fig 4. Diagram showing the processing life cycle in BIKE. All systems must have an input and output stream processor and one or more filters, one of which will usually be a `Classifier`.

Input Stream Processors

Input SPs are responsible for gathering a stream of cases from a source such as a file, database or directly from a user. Like other components in BIKE a new SP can be provided in a plugin to add required functionality. Current input SPs are:

- `C45InputStream` – used to load cases using the C4.5 [27] file format.
- `XMLInputStream` – used to load cases using the XML file format.
- `ODBCInputStream` – used to load cases and data directly from a system database.
- `W32UserInputStream` – used to load cases and data directly from a Windows user interface.

Output Stream Processors

Likewise Output SPs are responsible for supplying a stream of cases to a file, database or directly to a user. Like an input SP a new output SP can be provided in a plugin to add required functionality. Current output SPs provided with BIKE are:

- `C45OutputStream` – used to write cases to a C4.5 formatted file.
- `XMLOutputStream` – used to write cases to an XML formatted file.
- `W32UserOutputStream` – used to write cases to a Windows interface.

User Interfaces as Stream Processors

One interesting feature of BIKE is that there is no user interface directly implemented in the BIKE engine. Instead, a user interface in BIKE is developed as a type of input processing stream. In BIKE the input processing stream will first attempt to get the required details to form a case. A user interface SP simply gathers that case via the user interface and forwards it on to the next SP. One interesting feature of this is that SPs have the ability to send queries back to the SP that called them. Therefore, if an inference engine finds it is missing an attribute it requires, it can ask its calling SP. Each SP will then pass the request for the missing attribute back to its calling SP until one can resolve what the value should be. In a database SP this would be a query of the database, while in a user interface SP it will be a question displayed to the user. Likewise, a user output SP will display any response from the inference engine. The versatility of this approach is that we can attach any type of interface as an SP and it will automatically fit within the processing lifecycle of the engine. A second advantage is that any preprocessing of the attribute that is done, such as discretization, will be done automatically as the value is sent forward through the SPs. Currently the engine has implemented plugins for a console based input and output SP and a simple W32 based interface.

Filters

Filters are a little different in that they get and send their stream from and to another SP. Like other components a new filter can be provided in a plugin to add particular functionality required. Current filters provided with BIKE are:

- `Classifier` – Passes each case received to an inference engine for classification then forwards the combined case and result to the next SP. Allows the user to modify the knowledge base if the result is incorrect.
- `TestProcessor` – Passes each case received to an inference engine for classification then forwards the combined case and result to the next SP. Unlike the classifier it does not allow any corrections to be made to the knowledge base.
- `KFoldValidator` – This filter divides the input stream into equal sized pieces and forwards *k-1* to one filter and the k^{th} test fold to a second filter. It does this *k* times so that each fold has a turn at being the test fold. This is provided for general machine learning testing.

4.7 Virtual Expert

The Virtual Expert component with the base class `IExpert` provides a set of functions for getting a difference list from the inference engine and for creating rules through the selection of these differences. When a case has been processed by the inference engine the expert can indicate if the conclusion was incorrect. When this occurs, the virtual expert will be asked to correct the error by selecting differences in the difference list. In various implementations the inference engine may also ask other questions of the virtual expert. The provision of the virtual expert allows developers to extend and modify the way the virtual expert responds.

Currently the virtual expert has been extended by three implementations: console expert, W32 expert, and C4.5 expert. The console and W32 experts link to user interfaces that allow a human to provide the expertise to respond to the requests by the inference engine. The C4.5 based expert is a simulated expert based on those used in a number of RDR papers [28, 29]. This expert is used during a simulated training session, where each misclassification by the engine is corrected by selecting attributes from a decision tree generated using C4.5.

4.7 Plugins

The engine has a `PluginManager` that loads all plugins located in the plugin folder upon startup. Plugins must be located in this folder with the appropriate system extension for the system to find it. Upon being loaded, each plugin registers itself and the various components it extends with the `PluginManager`, giving each a unique key string. Once a plugin is registered, a client program can request to use the facilities provided by the plugin by giving the matching key. Currently BIKE provides extension points for new stream processors (input, output streams and filters), algorithms, knowledge representation schemes, knowledge base storage, classification schemes, virtual experts, types, functions and operators.

5 Conclusion and Future Work

This paper has introduced the Ballarat Incremental Knowledge Engine (BIKE) and provided an overview of the design and functionality of the engine. The engine contains four core components: the engine core, knowledge base, stream processors and virtual expert. Also discussed is the multitude of plugins that have also been developed to extend the engines functionality. This development, however, is far from complete. It is expected that over time this engine can be continuously expanded and improved.

This project has now been released to researchers and developers to add additional plugins to extend its functionality. If the reader wishes to take part in adding their work to the BIKE platform they should visit the BIKE website at http://bike.ballarat.edu.au/[5]. Currently the future plans for the engine are to improve the functionality of the current W32 interface by adding visualization tools, as well as to create a web based interface for visitors to experiment with. We also plan to develop more of the commonly used RDR algorithms along with additional SPs.

The RDR approach has long been recognized as a powerful approach to developing maintainable knowledge based system, however until now its take up has been limited by the lack of an extendable and open implementation. This release of BIKE will have benefits both to researchers, as well as software developers. This will allow a much wider take up of the RDR family of methodologies and facilitate the further advancement of the branch of research.

[5] The website will be made available in August as the server is still being configured.

Acknowledgements

This project was funded by the University of Ballarat through a Research Infrastructure Block Grant (RIBG) in 2009.

References

1. Compton, P., and Jansen, R., Knowledge in Context: a strategy for expert system maintenance, *Second Australian Joint Artificial Intelligence Conference (AI88)*, Vol. 1, pp. 292-306, 1988.
2. Richards, D. Two decades of Ripple Down Rules research, The Knowledge Engineering Review, 24 : 159-184 Cambridge University Press, 2009.
3. Compton, P., Edwards, G., Kang, B., Lazarus, L., Malor, R., Menziès, T., Preston, P., Srinivasan, A. and Sammut, C. Ripple Down Rules: Possibilities and Limitations. *6th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW91)*, Canada, SRDG publications, 1991.
4. Compton, P., Kang, B., Preston, P. and Mulholland, M. Knowledge Acquisition without Analysis. Knowledge Acquisition for Knowledge Based Systems, Berlin, Springer Verlag, 1993.
5. Menzies, T. Towards Situated Knowledge Acquisition. International Journal of Human-Computer Studies 49: 867-893, 1998.
6. Dazeley, R., and Kang, B. H. Epistemological Approach to the Process of Practice, Journal of Minds and Machines, Springer Science+Business Media B.V., 18: 547-567, 2008.
7. Dazeley, R. An Expert System Methodology for SMEs and NPOs, *11th Annual Australian Conference on Knowledge Management and Intelligent Decision Support - ACKMIDS08*, 2008.
8. Kang, B. H. and Compton, P. Multiple Classification Ripple Down Rules. *Third Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop*, Hatoyama, Japan, Japanese Society for Artificial Intelligence, 1994.
9. Kang, B. H. Validating Knowledge Acquisition: Multiple Classification Ripple Down Rules. Sydney, University of New South Wales, 1996.
10. Beydoun, G. and Hoffmann, A. NRDR for the Acquisition of Search Knowledge. In *Proceedings of Tenth Australian Joint Conference On Artificial Intelligence*, Perth, Australia, 1997.
11. Preston, P., Edwards, G., Compton, P., and Litkouthi, D. An Expert System Interpreter for Time Course Data with Refinement in Context, *AAAI Spring Symposium: Artificial Intelligence in Medicine*, 1994.
12. Shiraz, G. M. and Summut, C. A. An incremental Method for Learning to Control Dynamic Systems. *The Machine Learning Workshop of the IJCAI-95*, Montreal, Canada, 1995.
13. Martinez-Bejar, R., Benjamins, V., Compton, P., Preston, P. and Martin-Rubio, F. A formal framework to build domain knowledge ontologies for ripple-down rules-based systems. *11th Banff Knowledge Acquisition for Knowledge Base System Workshop (KAW98)*, Canada, SRDG, 1998.
14. Richards, D. Ripple Down Rules with Formal Concept Analysis: A Comparison to Personal Construct Psychology. *11th Workshop on Knowledge Acquisition, Modeling and Management (KAW'98)*, Banff, Canada, SRDG Publications, Department of Computer Science, University of Calgary, Calgary, Canada, 1998.

15. Vazey, M. and Richards, D. Achieving rapid knowledge acquisition in a high-volume call centre. In *Proceedings of the Pacific Knowledge Acquisition Workshop 2004*, Kang, B., Hoffmann, A., Yamaguchi, T. and Yeap, W. (eds) Auckland, 74-86, 2004.
16. Dazeley, R. and Kang, B. Rated MCRDR: Finding non-Linear Relationships between Classifications in MCRDR. in *3rd International Conference on Hybrid Intelligent Systems*. Melbourne, Australia: IOS Press, 499-508, 2003.
17. Dazeley, R., and Kang, B. H., Generalising Symbolic Knowledge in Online Classification and Prediction, Knowledge Acquisition: Approaches, Algorithms and Applications, Lecture Notes in Computer Science, Springer, Berlin, 5465: 91-108, 2009.
18. Compton, P., Peters, L., Edwards, G. and Lavers, T. Experience with ripple-down rules, In *Proceedings of AI-2005, the Twenty-Fifth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, Cambridge, UK, December, pp. 109-121, 2005.
19. Park, S. S., Kim, Y. S., Kang, B. Personalized Web Document Classification using MCRDR. In *Pacific Rim Knowledge Acquisition Workshop (PKAW04)*, Auckland, New Zealand, Springer, 2004.
20. Ho, V., Wobcke, W. and Compton, P. EMMA: an E-mail Management Assistant. In *IEEE/WIC International Conference on Intelligent Agent Technology*, Liu, J., Faltings, B., Zhong, N., Lu, R., and Nishida, T. (eds). IEEE, Los Alamitos, CA, 67-74, 2003.
21. Mak, P., Kang, B., Sammut, C. And Kadous, W. Knowledge acquisition module for conversational agents. In *Proceedings of the Pacific Knowledge Acquisition Workshop PKAW'04*, Kang, B., Hoffmann, A., Yamaguchi, T. and Yeap, W. (eds), Auckland, 54-62, 2004.
22. Finlayson, A. and Compton, P. Incremental knowledge acquisition using RDR for soccer simulation. In *Proceedings of the Pacific Knowledge Acquisition Workshop PKAW'04*, Kang, B., Hoffmann, A., Yamaguchi, T. and Yeap, W. (eds), Auckland, 102-116, 2004.
23. Gaines, B. R. and Compton, P. J. Induction of Ripple Down Rules. *Fifth Australian Conference on Artificial Intelligence (AI92)*, Hobart, World Scientific, 1992.
24. Scheffer, T. Algebraic Foundation and Improved Methods of Induction of Ripple Down Rules. In *Proceedings of the Pacific Knowledge Acquisition Workshop* (PKAW'96), 1996.
25. Edwards, G., Compton, P., Malor, R., Srinivasan, A. and Lazarus, L. Peirs: A pathologist-maintained expert system for the interpretation of chemical pathology reports, Pathology, Vol 25, No: 1, pp 27-34, 1993.
26. Garsden, H., Basilakis, J., Celler, B., Huynh, K. and Lovell, N. A Home Health Monitoring System Including Intelligent Reporting and Alerts. *EMBC 04: Annual Conference of the Engineering in Medicine and Biology Society*, San Francisco, CA, 2004.
27. Quinlan, J. R. C4.5: Programs for Machine Learning. San Mateo, California, Morgan Kaufmann Publishers, 1993.
28. Compton, P. Simulating Expertise. In *Proceedings of the 6th Pacific Knowledge Acquisition Workshop*, Sydney, Australia, 2000.
29. Dazeley, R., and Kang, B., Detecting the Knowledge Boundary with Prudence Analysis, In the *21st Australasian Joint Conference on Artificial Intelligence - AI-08* Auckland, New Zealand, Springer LNAI 5360, pp 482 – 488, 2008.