社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS2004-45 (2005-01)

Ripple Down Rule を用いたソースコード変更の分類

エディ パーキンソン† 川口 真司† 井上 克郎†

†大阪大学大学院情報科学研究科 〒560-8531 大阪府豊中市待兼山町1番3号
E-mail: †{eddy, s-kawagt, inoue}@ist.osaka-u.ac.jp

あらまし 開発者は時折大量のコード変更をせまられる．このようなソースコード変更はなるべく少ないほうが望ましい．Ripple Down Rule (RDR) は実績のある知識獲得手法である．RDR は経験から得られる知識をシンプルに管理するための手法であり，一度判定に失敗した事例を再利用することでより適切な判定を下すことを可能にする．本研究では，推薦されたコードの書き方と，実際に書きなおされたコードを確かめることで，RDR を用いて変更履歴を分類できるかどうかを検証した．まだより多くの作業が必要だが，得られた結果は RDR による分類が有用であることを示唆している．

キーワード Ripple Down Rules, Personal Software Process, Software Defects

Ripple Down Rules, a practical method of learning from code rewrites

Eddy PARKINSON† Shinji KAWAGUCHI† Katsuro INOUE†

†Department of Computer Science, Osaka University 1-3, Machikaneyama-cho, Osaka, 560-8531 Japan
E-mail: †{eddy, s-kawagt, inoue}@ist.osaka-u.ac.jp

Abstract Software developers sometimes rewrite large sections of program code. Reducing the number of rewrites would save valuable development time. Ripple Down Rules (RDR) has a proven knowledge acquisition track record. RDR looks to offer a simple to maintain method capturing knowledge gained through experience. RDR allows recommendations identified when a failure occurs, to be captured and reused. The approach is evaluated by examining recommended ways of writing code and examining rewritten code to see if modifications can be categorised using RDR. The results suggest the idea is feasible, although more work is needed.

Keyword Ripple Down Rules, Personal Software Process, Software Defects

1. Introduction

Software development suffers from budget overruns and project failures. A report by [StandishGroup01] states 20% of software development projects are cancelled and 50% of projects have an average cost of 189% of the original estimate. Personal software process (PSP), see [Humphrey95], has successfully achieved dramatic improvements in controlling budget overrun and improving the quality of the software delivered to customers, see [Ferguson99] for a case study. PSP is a gradual learning process that focuses on developers keeping notes to help them learn from previous mistakes. The idea described here is to combine PSP with Multiple Classification Ripple Down Rules. MCRDR helps novices learn from the choices of experts in a systematic way, see [Preston93] for a case study.

2. Understanding MCRDR

MCRDR works on the principle there are exceptions to every rule. When a failure occurs and some recommendation is made to help prevent such failures, the recommended action is added to the rule base. The recommended action exists to prevent a particular failure and only applies in certain situations. MCRDR is an extended version of RDR.

Figures 1 is a simple example used here to describe

Key: *Attribute* – **Conclusion**

- *Normal day* - **Wear shorts and t-shirt**
  - ➢ *Raining* - **Wear coat**
    - ✧ *Hot* - **T-shirt, shorts and take umbrella**
  - ➢ *Windy* - **Take kite**

Case:
Case1: Hot and sunny
Case2: Cold and raining
Case4: Hot and raining
Case3: Windy and sunny

Figure 1

RDR and MCRDR. On the first day, the first line "normal day" is created because on the first day it was hot and sunny so we decided to wear shorts and a t-shirt. When it is cold and raining, wearing shorts and t-shirt is not a good solution. The indentation indicates an exception to the rule, so the second day "cold and raining" is an exception, needing a different conclusion. Because wearing shorts and a t-shirt was not a good solution, the exception, "raining - wear coat" is added. To add the exception a difference between the first "Normal day" and the second day is needed. Either raining or cold can be used as an attribute to distinguish between the two cases. When picking attributes an expert tends to perform better than a novice. Experts tend to be better at picking attributes to distinguish between cases and as a result end up with a more compact set of rules, see [Kang95]. Experts are able to define complex attributes that distinguish between cases. Also by examining attributes picked by novices it has been possible to identify incorrect assumptions made by the novice.

Key: *Attribute* – **Conclusion**

- *Normal day* - **Wear shorts and t-shirt**
  - ➢ *Raining* - **T-shirt, shorts and take umbrella**
    - ✧ *Cold* - **Wear coat**
  - ➢ *Windy* - **Take kite**

Case:
Case1: hot and sunny
Case2: Hot and raining
Case4: Cold and raining
Case3: Windy and sunny

Figure 2

On the third day it is windy, hot and sunny, so the first conclusion is possible, but instead a new exception is added, "*Windy* - **Take kite**". By adding this exception the next time it is windy the conclusion take kite will be used, this is because there are no exceptions to the "*Windy* - **Take kite**" rule in the rule base.

Next a hot and raining day is encountered, because the rule "*Raining* - **Wear coat**" evaluates to true for a hot and raining day, an exception to this conclusion is added, "*Hot* - **T-shirt, shorts and take umbrella**". The structure is such than any rule can have an exception and the exception is itself a rule. Each rule is made up of an attribute condition and a conclusion, as can be seen in Figure 1.

The MCRDR approach allows several conclusions to be true. For example on a raining and windy day both "*Raining* - **Wear coat**" and "*Windy* - **Take kite**" are evaluated as being true. When an exception is true it means the parent conclusions are accepted as being false, and so when "*Hot* - **T-shirt, shorts and take umbrella**" is true, "*Raining* - **Wear coat**" is evaluated as false. RDR only allowed one conclusion in the tree to be true and so it was sometimes necessary to combine several conclusions into one, MCRDR solves this problems and has replaced RDR

Figure 2 shows that the order of the cases does not have a significant impact on knowledge acquisition. With the two rules, "*Raining* - **T-shirt, shorts and take umbrella**" and "*Cold* - **Wear coat**", in figure 2, it is still necessary to distinguish between hot and cold days and when it is raining and not raining. The same conclusions are reached and the same distinctions between cases also occur, just in a different order.

The simplicity comes from only having to identify a difference between two cases, rather than trying to create some complex problem solving method. When attempting to create problem solving methods [Compton98] notes that:

"we invariably attempted to develop something that was far more complex than is actually required."

The quote highlights the problem of developing simple rules that take into account exceptions. The rules in figures 1 & 2 show how MCRDR allows exceptions to be added to existing simple rules without restructuring.

3. Conclusion Classification with MCRDR

Medical experts have successfully used MCRDR to capture knowledge [Khan03], [Preston93]. When MCRDR was evaluated for use as a method for creating diets to fit patient preferences [Khan03], a senior dietician said the method offered "considerable potential to improve the daily routine". The work suggests problems that are suited to Case Based Reasoning (CBR) approaches are better tackled using MCRDR.

While the identification of conclusions can in many cases be automated, in some a partially manual search is needed. With a 1600 rule knowledge base such as [Preston93], a manual search of the rule base would be time consuming. It looks as if the time needed to find conclusions limits the scope of MCRDR. To help deal with this problem the MCRDR help desk system which [Kang97] describes contains a search mechanism to help search the large case base rather than relying on a manual search of the MCRDR tree. Another approach to the problem of searching through a large MCRDR case base was introduced by [Vazey04]. The method used is a sequence of requests for specific information, gradually refining the search for a conclusion.

4. Maintenance of MCRDR

One of the advantages of MCRDR over Expert Systems is that it does not require knowledge engineers to maintain the rule base, see [Khan03], [Preston93]. For an introduction to Expert Systems see [Jackson99]. The problem of maintaining an Expert System is that new knowledge is difficult to add because it tends to conflict with the existing rules in the system, this is highlighted in [Preston93]. In contrast to Expert Systems, MCRDR allows new knowledge without impacting on existing rule base. This advantage is highlighted by the quote:

"Rule addition of the order of 20 per hour could be achieved with very low error rates. It was realised that error rates could be eliminated by validating the rules as they were added." [Preston93]

5. Personal Software Process and Team Software Process (PSP & TSP)

PSP was developed by [Humphrey95], it has achieved impressive improvements in predicting schedules and reducing defects. The company technical report [Ferguson99] reported:

"Our average project schedule overrun has been reduced from 112% to 5%, and our average budget overrun from 87% to -4%."

This was achieved over several years and involved training all programmers in PSP. Several books were used in the training process and the training process looks to be central to PSP [Ferguson97]. Team Software Process (TSP) was developed by [Humphrey 99] after PSP to help with overall project management, see [Seshagiri03] for a case study.

PSP involves developers keeping track of defects they find and when they are removed, with the aim of trying to find ways of removing defects sooner in the development process. Methods such as requirements reviews and code reviews are employed to try and spot defects early and remove them long before unit testing and integration testing. PSP involves making developers aware of the types of defects that exist and the costs of removing them.

PSP & TSP also uses estimated LOC to help predict the number of defects that will exist. Such estimation techniques are continually refined by comparing actual LOC and actual defects to those estimated. The difference between the predicted and actual schedule and budget in [Ferguson99] had a standard deviation of about 25% for budget and 12% for the schedule spread over 14 projects.

6. Usefulness of combining MCRDR & PSP

Training is cited as a key part of PSP [Ferguson97]. The training includes the writing of 10 small programs and this case data is then used to train developers. The knowledge in MCRDR is also captured using case data. The advantage of using MCRDR is that it provides novices with the information they need to make expert like decisions right from the very beginning, rather than them having to relearn it from scratch.

While MCRDR does not claim to be a perfect method of passing on knowledge, it does appear to offer a better method of passing on expert knowledge than other approaches, see [Preston93], [Khan03]. Passing knowledge from an expert to a novice is difficult to do. This problem is highlighted by the following statement:

"[After] long term experience of maintaining an expert system. What became clear ... when an expert is asked how they reached a conclusion they do not and cannot explain how they reached their conclusion." see [Preston93].

To help deal with this problem RDR was developed, it uses actual case data to capture expert knowledge. PSP highlights the value of using past cases to guide the creation of schedules and aid the removal of defects.

Combining these two ideas lead to:

Hypothesis 1: Novices using MCRDR to predict budgets and schedules are able to achieve the same standard as experts in 95% of cases.

Hypothesis 2: Novices using MCRDR are able to be as effective at removing defects as experts in 95% of cases.

The 95% comes from the MCRDR literature, see [Preston93], and is an estimate of the percentage accuracy that MCRDR achieved in a domain. While the reality of what can actually be achieved is far from clear, the above hypothesises give a rough benchmark that can used to assess progress.

7. Evaluating code re-writes

The code modifications described in Figure 3 comes from sourceforge.net and appear to be typical of such projects. Even though only 18 cases are listed, many of the modifications are similar, which suggests MCRDR is able to separate modifications into simple categories. The process of using MCRDR to categorise modifications has highlighted two weaknesses:

1. As the rule base gets larger it becomes harder to add new rules, because much searching is needed to check if a similar case has been seen before.
2. The conclusions are labels that have little value; they describe modifications, but do not say what should be done to improve the situation.

The difficulty of searching the rule base means that there is a need for high reward conclusions to balance the time spent searching the rule base with the reward gained from implementing the conclusion. While several methods exist that speed up the searching of the rule base, these do not remove the issue of the time spent searching for conclusions. This suggests the conclusions added to the rule base would gain from having some measure of their time and quality value attached. Equally it means that having low value conclusions in the rule base could potentially have a negative impact on the value of the rule base. It may be possible to use some form of formal notation to help resolve the problem of searching the rule base, but this requires further investigation.

Because the conclusions are labels without a value, it makes it hard to decide if a conclusion is right or wrong. Until the categories reach the stage where the conclusions are useful in some way the choice of which categories are worth creating is arbitrary. This suggests that using MCRDR to categorise modifications provides little more than a rough list and the real advantage of using MCRDR will come when recommendations are created that improve things.

8. Conclusion

To test the idea of using MCRDR to pass software development knowledge from experts to novices there is a need to examine not only actual modifications made to code but also to identify cases where experts have been able to reduce the impact of project modifications. Identifying cases where an expert disagrees with a choice made by a novice should allow expert recommendations to be identified. Following this up by asking the expert to identify an attribute in the case that lead them to conclude an alternative action was needed should allow an MCRDR rule tree to be created.

The study of modified code shows that when classifying conclusions using MCRDR, the classification

Key: *Attribute–*	**Conclusion**	Case:
1. *Complete line(s) of code moved*	**statements that existed in more than one place are centralisation & enhanced**	1, 8
(ア) *Replaced variable/ constant with method*	**process used to produce a particular value type is centralised**	2
2. *Changed constant*	**simplistic name/text/visual enhancement or fix**	3, 5, 17
3. *Removed complete line(s) of code*	**feature had negative impact**	4, 14
4. *Apply function to many constants/variables*	**process to produce a particular value type is centralised**	6, 7, 10,11
5. *Modified a single section of code*	**Added simple feature/removed defect**	9, 15
(ア) *Inserted if + else statement around code*	**Added simple feature/removed bug**	14, 16
6. *Removed method call*	**Centralisation removed the need for the method call**	12, 13
7. *Inserted new lines of code*	**Added simple feature/removed bug**	14
8. *New methods inserted containing new code*	**Added feature**	15, 16

Figure 3 – MCRDR classification of code modification

has little value, when the conclusion is not useful.

The examination of the MCRDR and PSP literature and code re-writes shows that using MCRDR to pass on expert PSP knowledge to novices is a promising idea. What remains unclear is the detail of actual recommendations discovered while using PSP. Adding actual PSP recommendations to a MCRDR rule base should reveal more about the value and feasibility of the hypothesises described above.

9. References

[Compton98] A Trade Off Between Domain Knowledge and Problem Solving Method Power 、 Eleventh Workshop on Knowledge Acquisition, P. Compton, Z. Ramadan, P. Preston, T. Le-Gia, V.Chellen, M. Mulholland, D.B. Hibbert, P.R. Haddad, B. Kang 1998.

[Ferguson97] Results of Applying the Personal Software Process, IEEE Computer, Vol. 30, No. 5, P. Ferguson, W. Humphrey, S. Khajenoori, S. Macke, and A. Matvya 1997

[Ferguson99] Software Process Improvement Works, SEI Technical Report, CMU/SEI-TR-99-27 Ferguson P., Leman G., Perini P., Renner S. & Seshagiri G. 1999

[Humphrey95] A Discipline for Software Engineering, Addison-Wesley, Watts Humphrey 1995.

[Humphrey99] Introduction to the Team Software Process, Addison-Wesley, Watts Humphrey 1999

[Jackson99] Introduction to Expert Systems, 3rd ed., Addison-Wesley, Harlow, England, Peter Jackson 1999.

[Kang95] Multiple Classification Ripple Down Rules Evaluation and Possibilities, in Proceedings 9th Banff Workshop on Knowledge Acquisition, B.H. Kang, P. Compton, and P. Preston 1995

[Kang97] Help Desk System with Intelligent Interface, in Applied Artificial Intelligence, 11: 611-631, Kang, B. H., Yoshida, K., Motoda, H. and Compton, P. 1997

[Khan03] Building a case based diet recommendation system without a knowledge engineer, in Artificial Intelligence in Medicine 27(2): 155-179, Abdus Salam Khan, Achim G. Hoffmann 2003

[Preston93] A 1600 Rule Expert System Without Knowledge Engineers, in J. Leibowitz, editor, Second World Congress on Expert Systems, P. Preston, G. Edwards, and P. Compton 1993

[Seshagiri03] Walking the Talk Building Quality into the Software Quality Management Tool, Third International Conference On Quality Software, Girish V. Seshagiri, S. Priya 2003

[StandishGroup01] CHAOS Chronicles II, The Standish Group, www.standishgroup.com, 2001

[Vazey04] Achieving Rapid Knowledge Acquisition in a High-Volume Call Centre, in Proceedings of the Pacific Knowledge Acquisition Workshop, 2004