STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVIII**, Number 1, 2013

A STUDY ON ASSOCIATION RULE MINING BASED SOFTWARE DESIGN DEFECT DETECTION

ZSUZSANNA MARIAN

Abstract. In this paper we are investigating the effect of parameter variations for a method we have previously introduced for detecting software design defects. This method uses software metrics and relational association rules to find badly designed classes. We perform five different studies, to see the effect of using normalized or original software metric values, the effect of mining only binary or any-length rules, the effect of mining only maximal or all rules and the effect of changing the value of the minimum support for the rules. We are also investigating the changes caused by modifying the value of the parameter that determines which classes to report as having bad design.

1. Introduction

Software systems developed and used in our days are more and more complex, because the tasks they have to solve are getting more and more complex as well. In these systems both maintenance and the addition of new features is a complicated task, which is even more complicated if the system has a bad design. This is why different techniques exist that try to identify design defects in a system, in order to correct them, thus making maintenance easier.

In [3] we have presented a novel method for design defect detection, called *Software Design Defect detection using Relational Association Rules* or SDDRAR. This approach uses software metrics, which are often used for software design defect detection, but it also uses Relational Association Rules, a particular type of association rules, which were not used so far for this task.

We have provided six different open source case studies in [3], to show the effectiveness of our method. We have also presented some similar approaches form literature and compared the SDDRAR method to them. In this paper,

2010 *Mathematics Subject Classification.* 68P15, 68N99.

1998 *CR Categories and Descriptors.* H.2.8 [**Database Management**]: Database Applications - Data Mining; D.2.10 [**Software Engineering**]: Design;

Key words and phrases. Relational association rules, Software metrics, Design defect detection.

we intend to present a study performed with different parameter settings for our method. We want to investigate the following aspects:

- Using the original or normalized values for software metrics.
- Using binary relational association rules or using relational association rules of any length.
- Using only maximal relational association rules, or using all mined rules.
- The effect of modifications for the τ parameter and the minimum confidence on the prediction accuracy.

The rest of the paper is structured as follows: Section 2 presents the theoretical background for the SDDRAR approach, presenting Relational Association Rules (Section 2.1), the used software metrics (Section 2.2), the SDDRAR method (Section 2.3) and the open source projects used for testing (Section 2.4). Section 3 presents the studies performed with different variations of the parameters. Section 4 briefly presents some similar approaches and compares our method to them, while Section 5 concludes the paper.

2. Background

In this section we will present the main theoretical background of the SDDRAR method. A more detailed description was given in [3].

2.1. **Relational Association Rules.** Relational association rules are an extension of regular association rules and are able to capture different types of relationships between record attributes. They were introduced in [16], and can be formally defined in the following way: let $R = \{r_1, r_2, ..., r_n\}$ be a set of instances, where each instance is characterized by a vector of m attributes: $r_i = \{a_1, a_2, ..., a_m\}$. Each attribute a_i takes value from the domain D_i, and the value of an attribute a_i in an instance r_j is denoted by $\Phi(r_j, a_i)$. Between two domains D_i and D_j different relations can be defined, for example equal, less than, greater than, etc. We denote by M the set of all possible relations that can be defined on $D_i x D_j$.

Using the notations presented above, a relational association rule is of the form $(a_{i_1}, a_{i_2}, \ldots, a_{i_l}) \Rightarrow (a_{i_1}\ \mu_1\ a_{i_2}\ \mu_2\ a_{i_3}\ \ldots\ \mu_{l-1}\ a_{i_l})$ where $\{a_{i_1}, \ldots, a_{i_l}\} \subseteq A = \{a_1, a_2, ..., a_m\}$, $a_{i_j} \neq a_{i_k}$, $j, k = 1..l, j \neq k$ and $\mu_i \in M$ is a relation over $D_{i_j} \times D_{i_{j+1}}$, D_{i_j} being the domain of attribute a_{i_j}.

Like regular association rules, relational association rules are also characterized by two values, *support* and *confidence*, defined in the following way:

- If $a_{i_1}, a_{i_2}, \ldots, a_{i_l}$ occur together (are non-empty) in $s\%$ of the instances, we call s the *support* of the rule.

- If $R' \subseteq R$ is the set of instances where $a_{i_1}, a_{i_2}, \ldots, a_{i_l}$ occur together and $\Phi(r\prime, a_{i_1})\ \mu_1\ \Phi(r\prime, a_{i_2})\ \mu_2\ \Phi(r\prime, a_{i_3})\ \ldots\ \mu_{l-1}\ \Phi(r\prime, a_{i_l})$ is true for each instance $r\prime \in R'$, we call $c = \frac{|R'|}{|R|}$ the *confidence* of the rule.

Another value that characterizes a relational association rule is its *length*, which is the number of attributes in the rule. The minimum possible length for a rule is 2, while the maximum possible length is the number of attributes.

In a dataset many association rules can be found, so usually only those are mined which have support and confidence higher than some user specified threshold, s_{min} and c_{min}. These rules are also called *interesting*. In [3] we have introduced an A-Priori [1] like algorithm, called *DRAR*, which can find all the interesting relational association rules in a dataset. Also, the algorithm can be configured to find only the maximal interesting relational association rules (rules which cannot be further extended with an attribute, because they will no longer be interesting) in the dataset.

2.2. **Software Metrics.** In order to mine relational association rules from a dataset, one needs a set of instances, where each instance is actually a vector of attributes. In our model, the instances were classes of the software system, while the attributes that characterize these instances are the values of different software metrics. The set of software metrics is denoted by $SM = \{sm_1, sm_2, \ldots, sm_k\}$. So, we consider a software system S as being a set of instances (classes) $S = \{s_1, s_2, \ldots, s_n\}$, where each instance s_i is represented as a k-dimensional vector $s_i = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$, where s_{i_j} is the value of the software metric sm_j for the instance s_i.

We have identified in [10] a set of 16 software metrics that measure the size, cohesion and coupling in a software system. After performing some statistical analysis on this set in [3], after which we eliminated 4 of them, the set of software metrics became $SM = \{CBO, DAC, ICH, INS, LCC, LCOM1, LCOM2, LCOM4, LCOM5, LD, MPC, NOA\}$. A description of these metrics and the statistical analysis approach performed are presented in [3].

2.3. **The SDDRAR method.** The aim of the SDDRAR method is to identify classes with design defect in a software system, using relational association rules. It has three different steps, which will be presented briefly in this section.

- Data collection and pre-processing.
- Building the SDDRAR model
- Testing

In the first step, *Data collection and pre-processing*, a set S_{good} of well-designed software systems is collected. Next, the k-dimensional representation of the classes (entities) from these software systems is built, denoted by DS, using the

set of software metrics presented in Section 2.2. Then, the already mentioned statistical analysis on the set of initial software metrics is performed.

During the second step, *Building the SDDRAR model*, all interesting relational association rules of any length are discovered in the DS dataset. The relations used between the attributes are $\leq$ and $\geq$ and are not defined for metrics with the value 0. These rules, kept in a set called RAR, will be used to identify classes (entities) with bad design in other software systems.

The last step, *Testing*, is the most complex. In this step, a new software system, S_{new} has to be analyzed to detect those entities that have a bad design. First, the k-dimensional representation of the classes from S_{new} is built (using the same software metrics as above). Then, for each entity $e_i \in S_{new}$ the *number of errors*, $err(e_i)$, is computed, as the number of relational association rules from RAR that are not verified by the k-dimensional representation of the entity. Next, the *percentage of error*, $pe(e_i)$, is computed, as: $pe(e_i) = \frac{err(e_i)}{|RAR|}$. After this, the set of *potential errors*, P_τ, is determined, containing those entities, for which $pe(e_i)$ is greater than a user specified threshold, τ. We have two possible cases:

(1) If this set is empty, a threshold ϵ is computed as the sum between the mean and the standard deviation of the non-zero number of errors for all entities. The SDDRAR method will report as ill structured software entities in S_{new} the ones which have $err(e) > \epsilon$.
(2) If P_τ is not empty, than the average number of errors of the entities from P_τ, avg, is computed and the algorithm will report as ill structured software entities the ones, for which $err(e) > avg$.

2.4. **Open source projects used.** In this section we will shortly present those open source projects that were used in [3] and will be used in this paper, too. First, as presented above, the SDDRAR method needs a set of well designed software systems, S_{good}, in order to build the set of relational association rules. In our current implementation this set contains one single element, the *JHotDraw* [5] software system, built by Erich Gamma and Thomas Eggenschwiler. It is considered an example of good design and use of design patterns. It consists of 173 classes, out of which only 132 are used, because the rest are interfaces, for which the value of some metrics cannot be computed.

We have also used 6 different systems for testing our method: two simple artificial examples and 4 open source systems, taken from the SourceForge repository. For these 4, we have also considered consecutive versions, to see how reported classes and the number of errors change as the project evolves. These 4 systems are the following:

- FTP4J, [4], a Java implementation of a full-featured FTP client. 4 consecutive versions were considered, 1.5, 1.5.1, 1.6, 1.6.1, each having 27 classes (and 8 interfaces which were not included in the analysis).
- ISO8583, [6], an implementation of the ISO 8583 protocol in Java. Three versions were used, 1.5.2, 1.5.3, 1.5.4, each containing 21 classes (and 2 interfaces).
- Profiler4J, [15], a CPU profiler for Java, which supports remote profiling and on-the-fly configuration. From the two jar files of the project, we used four versions of the *agent.jar* file: 1.0-alpha5, 1.0-alpha6, 1.0-alpha7 and 1.0-beta1. The first two versions have 18 classes, while the last two have only 15.
- WinRun4J, [17], a Windows native launcher for Java applications. Five consecutive versions were analyzed, 0.4.0, 0.4.1, 0.4.2, 0.4.3 and 0.4.4. The first three versions have 21 classes, while the last two have 24.

All projects (including JHotDraw) were analyzed using the ASM 3.0 bytecode manipulation framework [2]. We use this framework to extract a representation of a software system containing classes, methods and attributes and the relations between them. This representation is then used to compute the value of the software metrics for the classes.

3. Study on the parameter variations

The SDDRAR method presented in Section 2.3 has many different parameters, whose value can influence the results. For example, the parameters of the relational association rule mining algorithm are the s_{min} and c_{min} values, but also whether the algorithm should find only binary rules or rules of any length, or whether all mined rules should be kept, or only the maximal ones. Related to the software metrics, one parameter is, whether the original or the normalized value of the metrics should be used for the relational association rule mining. And for the SDDRAR algorithm, the value of τ is very important, because this is the one that determines the reported classes. In this section we will present some experiments and analysis of the results, when changing the values of these parameters. The first three experiments were performed with $c_{min} = 0.85$ and $s_{min} = 0.9$.

3.1. **Normalized vs. Original software metrics values.**

The first test we are going to execute is to see how using normalized or non-normalized (called original) software metrics values influences the results of the algorithm.

After computing the values of the software metrics for the JHotDraw system (both normalized and original values), we ran our algorithm to extract the set of maximal relational association rules of any length. The number of

rules for each length and the total number of rules for both cases can be seen in the first two columns of Table 1.

TABLE 1. Number of rules for different rule mining settings for the jHotDraw system.

Length	Normalized, Any-length, Maximal	Original	Binary	All-mined
2	0	0	14	14
3	21	5	0	39
4	4	6	0	22
5	16	160	0	18
6	1	0	0	1
Total	42	171	14	94

We have also divided the relational association rules into simple binary rules, to see which pairs make up the longer rules. The exact pairs are omitted because of lack of space, but analyzing them, we observed that there is only one rule which is common ($LCOM5 \leq LD$), but there are six rules which appear with one relation when using the original values, and appear with the inverse relation, when normalized values are used. Also, the presence of some of the rules for the original metric values is easy to understand. The 12 software metrics used can be divided into two categories: metrics whose value is between 0 and 1 (ICH, INS, LCC, LCOM5 and LD), and metrics whose value can be greater than 1 (CBO, DAC, LCOM1, LCOM2, LCOM4, MPC and NOA). When the metric values are not normalized, it is logical to have many cases when the value of a metric from the first category is less than the value of a metric from the second category. Indeed, if we check the rules for the original values, we can see that for 12 out of the 19 rules this is the case, while for normalized values there is no such rule. This shows that the relational association rules found for the original values uncover a lot less hidden patterns in data than the normalized values.

The mined rules were used to find badly designed classes in the FTP4J software system, version 1.5, with $\tau = 0.8$. The reported classes are presented in the first two columns of Table 2. We can see, that the two variants report very different results. As presented in [3], the *FTPClient* class should be reported, because it is a God Class, having above 3500 lines of code (with comments), 34 attributes (second highest value in the system being 7) and 84 methods (second highest is 15). If we verify the classes reported when using the original metric values, we can see that classes *Base64OutputStream*, *SOCKS4Connector* and *SOCKS5Connector* should not be reported, because

they are not badly designed. The reason why they are reported is that they use no other class from the system, so their CBO and MPC values are 0, causing a lot of errors. The class *FTPFile* can be considered a Data Class (5 attributes, and 11 methods - getters, setters and toString), but most of its errors are because of the CBO and MPC metrics, just like in case of the previous classes, suggesting that there is no pattern in errors for finding Data Classes (and nor for God Classes, because FTPClient was not identified either).

TABLE 2. Classes reported by the SDDRAR method for the FTP4J system for different parameter settings.

Normalized Any-length Maximal	Original	Binary	All-mined
FTPClient	FTPFile Base64OutputStream SOCKS4Connector SOCKS5Connection	FTPClient	FTPClient DirectConnector FTPDataTransfer- Exception

Considering the above presented case study, we can conclude, that it is better to use normalized software metric values, both because they predict bad designs better, and because they can find hidden patterns in the metric values better.

3.2. **Binary vs. Any-length relational association rules.** In this section we will investigate the effect of using only binary, or any-length rules. First, for the *JHotDraw* system, we have generated all maximal rules, once only the binary ones, and after that rules of any length.The number of rules for each length and the total number of rules for both cases can be seen on the first and third column of Table 1.

Dividing the any-length rules in binary pairs, we get the exact same pairs as for the normalized values in the previous Section (which is normal, because they were mined from the same data). They are also the same rules mined when only binary rules are considered. When using these rules to identify badly designed classes in the FTP4J software system, version 1.5 and $\tau = 0.8$, in both cases a single class is identified: *FTPClient* (as it is presented in the first and third column of Table 2). The difference is that when the binary rules are used, case 1 from 2.3 has to be used ($P_{0.8} = \emptyset$). Although the results are the same, both for binary and any-length relational association rules, we think that using longer association rules is better, because the fact that a binary relation can appear more than once in the set of rules, provides kind of a weighting mechanism, making binary relations that appear more than once,

more important. For example, out of those classes from FTP4J which have one error when using the binary rules, some have 2 errors when using longer rules, while others have 9, meaning that they broke relations, which appear in more rules, so they can be considered more important. So, we conclude that it is better to use longer relational association rules.

3.3. **Maximal vs. All-mined relational association rules.** In this section we will investigate the effect of using only the maximal relational association rules, or using all the mined rules, even if later they were extended with other metrics. The number of such rules in the *jHotDraw* system is presented in the last column of Table 1.

The results of using these rules for finding badly designed classes in the *FTP4J* software system, version 1.5, with $\tau = 0.8$ is presented in the last column of Table 2. In case of the all-mined rules, $P_{0.8} = \emptyset$, so case 1 from 2.3 is used.

As already presented above, the identification of the *FTPClient* class is justified, because it is a God Class. The other two classes identified using all the mined rules have the exact same errors, for the CBO, INS, LCOM1 and LCOM4 metrics. Verifying the source code, we can see that these classes are simple classes with 3 or 4 very short (one line) methods, but no design defect can be found in them. So we can conclude that using only maximal rules is better.

3.4. **Changing the value of τ.** In the previous three Sections, we have conducted three different studies, and concluded that it is better to use normalized software metric values for mining relational association rules, and that mining any-length but only maximal rules gives better results. In this Section we will investigate the effect of changing the value of τ, the parameter which determines which classes to report as badly designed. Also, instead of using only one system, we are going to use all four software systems presented in 2.4 with all their versions.

It is expected that lowering the value of τ will result in more classes being reported as having errors, while increasing it will lead to less reported classes. Although, if τ is too high, it might happen that $P_\tau = \emptyset$ (case 1 from 2.3), which can lead to a situation, when an increased τ leads to more reported classes. After this point, increasing the value of τ will not give different results.

During the experiments we varied the value of τ from 0.2 to $0.8 - 0.95$ (depending when we got to the point when there was no use increasing it anymore) with increments of 0.05. The results (identified classes) are presented on Tables 3, 4, 5 and 6. A star after the name of an identified class means that it was identified using case 1 from 2.3. Cases when the result did not change

for more consecutive values of τ are presented with the interval for which the values are true.

TABLE 3. Identified classes for the *FTP4J* project for different values of τ.

τ	1.5, 1.5.1, 1.6, 1.6.1
0.9	*FTPClient**
0.85 - 0.25	*FTPClient*
0.2	*FTPClient* *DirectConnector* *FTPDataTransferException*

From Table 3 we can see that the results are quite stable, the value of τ has to be decreased until 0.2 in order to report classes *DirectConnector* and *FTPDataTransferException*, which, as presented in Section 3.3, are not badly designed.

TABLE 4. Identified classes for the *ISO8583* project for different values of τ.

τ	1.5.2, 1.5.3, 1.5.4
0.95	*MessageFactory**
0.9 - 0.65	*MessageFactory*
0.6 - 02	*MessageFactory* *ISOValue*

From Table 4 we can see that when decreasing the value of τ to 0.6 (and lower) a new class is reported, *ISOValue*. Although this class has some minor design defects (for example calling the getters in the *equals* method, instead of using the fields directly) there are no big problems with it.

Table 5 presents the classes reported for the *Profiler4J* project. In [3] we have presented that the classes reported for $\tau = 0.8$ (*Server* for version 1.0-alpha5, *MemoryInfo* for version 1.0-alpha6 and *Config* for versions 1.0-alpha7 and 1.0-beta1) are justified. *Server* has too high coupling, *MemoryInfo* is a Data Class (which actually disappears after version 1.0-alpha6) and *Config* can be considered a Data Class, too. Besides these three classes, only *ThreadInfo* is reported, but only for low values of τ. Checking the source code, we can see that the class has many static methods, but does not really have errors.

Table 6 presents the classes reported for the WinRun4J system. In [3] we considered the *NativeBinder* class as having smaller design defects (a too long

TABLE 5. Identified classes for the *Profiler4J* project for different values of τ

.

τ	1.0-alpha5	1.0-alpha6	1.0-alpha7, 1.0-beta1
0.9	*Server** *MemoryInfo**	*Config** *MemoryInfo**	*Config**
0.85	*Server*	*Config** *MemoryInfo**	*Config**
0.8 - 0.75	*Server*	*MemoryInfo*	*Config*
0.7 - 0.6	*Server* *MemoryInfo*	*MemoryInfo*	*Config*
0.55 - 0.5	*Server* *MemoryInfo*	*Config* *MemoryInfo*	*Config*
0.45	*Config* *Server* *MemoryInfo*	*Config* *MemoryInfo*	*Config*
0.4 - 0.3	*Config* *Server* *ThreadInfo* *MemoryInfo*	*Config* *MemoryInfo*	*Config* *ThreadInfo*
0.25 - 0.2	*Config* *Server* *ThreadInfo* *MemoryInfo*	*Config* *Server* *ThreadInfo* *MemoryInfo*	*Config* *ThreadInfo*

method, high coupling, etc.), instead of having just one main problem. For $\tau = 0.6$ and less, the class *Launcher* is also reported. It is a class with many methods, and many overloaded methods, which call each other, as suggested by the errors for the ICH metric. High value of LCOM1 metric (and many errors related to this metric) suggest that the class is not really cohesive. Another reported class is *Closure*, but besides having a really long method, this class is fine. Finally, the *FileVerb* class reported for $\tau = 0.45$ and lower, is a Data Class.

Considering the results for the above presented four systems, we can say that small differences in the value of the τ parameter will not result in big changes in the classes. As we expected, lowering the value of τ will result in more classes reported, but not a lot more. In [3] we have shown that the results given for $\tau = 0.8$ are correct (those classes indeed have problems), and now we presented that most of the other classes reported (for lower values of

TABLE 6. Identified classes for the *WinRun4J* project for different values of τ.

τ	0.4.0 - 0.4.4
0.8 - 0.75	*NativeBinder**
0.7 - 0.65	*NativeBinder*
0.6 - 0.55	*NativeBinder* *Launcher*
0.5	*Closure* *NativeBinder* *Launcher*
0.45 - 0.2	*Closure* *NativeBinder* *FileVerb* *Launcher*

τ) also have design problems to a given extent. This suggests that if there is sufficient time for an analysis, it might be worth lowering the value of τ to get not only the class with the biggest problems, but also other classes with smaller ones.

3.5. **Changing the value of c_{min}.** .

In this section we are going to investigate the effect of changing the value of the minimum confidence (so far we have performed every test with the value of 0.85). While changes in the value of τ influence which classes are reported as having a bad design, but each class had the same number of errors (as shown in the previous section), changes in the value of c_{min} influence which rules are mined, and consequently the number of errors for each class. For these experiments we will use the value of $\tau = 0.8$.

First we wanted to analyze how the number and length of rules changes when the value of c_{min} is increased or decreased. For values between 0.9 and 0.6 with decrements of 0.05 the number of rules is presented in Table 7. We can see that as the minimum confidence decreases, the number of mined rules increases drastically. The maximum length of the rules increases, too.

The results of using the relational association rules mined for the different values of c_{min} for finding badly designed classes in the open source projects are presented on Tables 8, 9, 10, 11. Looking at the tables, we can see that as the number of rules increases, so does the number of reported classes.

In case of the *FTP4J* project (presented on Table 8), besides the well-justified *FTPClient* class, four other classes are reported. Out of these, as

TABLE 7. The number and length of rules for different values of c_{min}.

Confidence	3	4	5	6	7	8	Total
0.9	21	0	0	0	0	0	21
0.85	21	4	16	1	0	0	42
0.8	7	3	132	110	0	0	252
0.75	4	7	113	208	60	0	392
0.7	0	4	132	303	207	30	676
0.65	20	16	111	308	443	146	1044
0.6	25	22	127	506	696	286	1662

TABLE 8. Identified classes for the *FTP4J* system for different values of c_{min}.

c_{min}	1.5 - 1.5.1	1.6 - 1.6.1
0.9 - 0.8	*FTPClient*	*FTPClient*
0.75	*FTPClient* *SOCKS4Connector* *SOCKS5Connector*	*FTPClient* *SOCKS4Connector* *SOCKS5Connector*
0.7	*FTPClient* *DirectConnector* *SOCKS4Connector* *SOCKS5Connector*	*FTPClient* *DirectConnector* *SOCKS4Connector* *SOCKS5Connector*
0.65-0.6	*FTPClient* *DirectConnector* *FTPFile* *SOCKS4Connector* *SOCKS5Connector*	*FTPClient* *FTPFile* *SOCKS4Connector* *SOCKS5Connector*

described in previous Sections, only the class *FTPFile* is reported correctly, because it is a Data Class.

The results given for the *ISO8583* project are really interesting. For the other projects, a class which is reported for a higher value of c_{min} will be reported for the lower values, too (there is one exception for the *FTP4J* project, and one for *Profiler4J*), but here classes keep appearing and disappearing. There are three classes which are reported: *MessageFactory*, *ISOValue* and *ISOType*. We have already presented that *MessageFactory* is correctly identified, but *ISOValue* is not. *ISOType* is actually an enum, with many methods, many of them overloaded, but there are no outstanding problems with it.

TABLE 9. Identified classes for the *ISO8583* system for different values of c_{min}.

c_{min}	1.5.2	1.5.3 - 1.5.4
0.9 - 0.85	*MessageFactory*	*MessageFactory*
0.8 - 0.75	*ISOValue* *MessageFactory*	*ISOValue* *MessageFactory*
0.7	*ISOValue* *ISOType*	*ISOValue* *ISOType*
0.65	*ISOValue* *ISOType* *MessageFactory*	*ISOValue* *ISOType* *MessageFactory*
0.6	*ISOValue* *MessageFactory*	*ISOValue* *ISOType* *MessageFactory*

TABLE 10. Identified classes for the *Profiler4J* system for different values of c_{min}.

c_{min}	1.0-alpha5	1.0-alpha6	1.0-alpha7 1.0-beta1
0.9	*Server*	*MemoryInfo**	*Config**
0.85	*Server*	*MemoryInfo*	*Config*
0.8	*Config* *Server* *MemoryInfo*	*Config* *MemoryInfo*	*Config*
0.75	*Config* *Server* *MemoryInfo* *Response*	*Config* *CFlow* *MemoryInfo* *Response*	*Config* *CFlow*
0.7-0.6	*Config* *Server* *ThreadInfo* *MemoryInfo* *Response*	*Config* *ThreadInfo* *MemoryInfo* *Response*	*Config* *ThreadInfo* *CFlow*

The reason for the fact that some reported classes disappear when the value of c_{min} decreases (and sometimes they appear again) is caused by the fact, that for different values of c_{min} the percentage of binary rules in which a metric appears can change a lot. For example, when $c_{min} = 0.85$, the LD

TABLE 11. Identified classes for the *WinRun4J* system for different values of c_{min}.

c_{min}	0.4.0 - 0.4.4
0.9	*NativeBinder*
0.85	*NativeBinder**
0.8	*NativeBinder* *Launcher*
0.75 - 0.6	*NativeBinder* *FileVerb* *Launcher*

metric appears in 29.26% of the binary rules, while for $c_{min} = 0.6$ this value is only 17.56%. Because of these changes the order of classes based on their number of errors changes too. For example, for version 1.5.2 , for $c_{min} = 0.9$, *MessageFactory* has more errors than *ISOValue* and *ISOType* together. When $c_{min} = 0.8$, *ISOValue* has more errors than *MessageFactory*, and *ISOType* has the least (out of these three). When $c_{min} = 0.65$, *MessageFactory* is the one with the least errors and *ISOValue* has the most. Finally, when $c_{min} = 0.6$, *ISOValue* has again the most, but now *ISOType* has the least errors.

The results for the *Profiler4J* software system are presented on Table 10. Here we also have some classes that were not reported during the previous studies: *Response* and *CFlow*. *Response* is a simple Data Class, with two fields and two getters, while *CFlow* is a class with only one short method and an inner class (which is not included in our analysis).

For the *WinRun4J* project, the reported classes are almost the same as in the previous Section, when the value of τ was changed. As already mentioned there, *FileVerb* is a Data Class andd *Launcher* is not really cohesive.

Verifying the results for all software systems, we can observe that for the lower values of c_{min} (between 0.7-0.6) three out of the four projects report new Data Classes: *FileVerb* in case of *WinRun4J*, *FTPFile* in case of *FTP4J* and *Response* in case of *ISO8583*. This suggests that further analysis might be useful, to see, if it is possible to identify a value or an interval for c_{min}, where it can identify Data Classes in the system.

4. Comparison to Related Work

There are many different approaches that try to identify design defects in software systems, presented in the literature. Most of them use software metrics and predefined thresholds for their values, like Marinescu's *detection strategies* [9], or Munro's method, presented in [14]. In a series of papers,

Moha et. al presents the idea of *rule cards*, which contains both metric values, but also semantic and structural information [12], [11], [13]. Rule cards were extended by Khomh et. al, in order to handle uncertainty [8]. They use Bayesian Belief Networks and assign to each class a probability that it contains a given design defect. In [7] three different search techniques (Harmony Search, Particle Swarm Optimization and Simulated Annealing) are used for finding rules that describe design defects, and can later be applied to classes. These rules are made of software metrics and threshold values for them.

Just like the above presented methods, our approach uses software metrics, but it also uses Relational Association Rules, which, according to our knowledge, have never been used for design defect detection yet. Another important difference between them, is that our method uses the relation between the values of different metrics, while the above presented ones use fixed thresholds, which can be hard to determine, because "good" and "bad" values for a metric usually depend on the size of the software system. On the other hand, the above presented methods are capable of identifying different, well-defined software smells, like Data Class or God Class, while our method can only identify the class with the design problem.

5. Conclusions

In this paper we have presented a study on the effect of changes for different parameters for the SDDRAR method, an approach for detecting design defects in software systems, using software metrics and relational association rules. We have considered five different possible changes, and reported and analysed the results on open source software systems. We showed that it is better to use normalized software metric values for mining relational association rules, that it is better to use any-length association rules (not just binary) and that using maximal rules is better. We have also shown that the parameter values used in [3] for c_{min} and τ (0.85 and 0.8, respectively) are good values, but interesting results could be achieved with different values, too.

The last study performed (when consequences of changes in the value of c_{min} were tested) could be further analyzed. It would also be worth investigating how the binary rules and their number change for different values of c_{min}. We have already identified a possible pattern, that for lower values Data classes are found, but this should be tested.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[2] ObjectWeb: Open Source Middleware, 2012. http://asm.objectweb.org/.
[3] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Detecting software design defects using relational association rule mining. *Knowledge and Information Systems*, 2012. Under review.
[4] Ftp4j, 2012. http://sourceforge.net/projects/ftp4j/.
[5] E. Gamma. JHotDraw Project, 2012. http://sourceforge.net/projects/jhotdraw.
[6] Iso8583, 2012. http://sourceforge.net/projects/j8583/.
[7] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer. Search-based design defects detection by example. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*, pages 401–415, 2011.
[8] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software*, pages 305–314, 2009.
[9] Radu Ma-rinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Politechnica University Timisoara, Faculty of Automatics and Computer Science, 2002.
[10] Zsuzsanna Marian. Aggregated metrics guided software restructuring. In *Proceedings of the 8th IEEE International Conference on Intelligent Computer Communication and Processing*, pages 259–266, 2012.
[11] Naouel Moha. Detection and correction of design defects in object-oriented architectures. In *Doctoral Symposium, 20th edition of the European Conference on Object-Oriented Programming*, 2006.
[12] Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 297–300, 2006.
[13] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22(3–4):345–361, 2010.
[14] Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source code. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, 2005.
[15] Profiler4j, 2012. http://sourceforge.net/projects/profiler4j/.
[16] Gabriela Serban, Alina Câmpan, and Istvan Gergely Czibula. A programming interface for finding relational association rules. *International Journal of Computers, Communications & Control*, I(S.):439–444, June 2006.
[17] Winrun4j, 2012. http://sourceforge.net/projects/winrun4j/.

Department of Computer Science, Babeş-Bolyai University, 1 M. Kogălniceanu St., 400084 Cluj-Napoca, Romania
E-mail address: marianzsu@cs.ubbcluj.ro