Supervised Software Modularisation

Mathew Hall
Department of Computer Science
The University of Sheffield
UK
m.hall@dcs.shef.ac.uk

Neil Walkinshaw
Department of Computer Science
The University of Leicester
UK
nw91@leicester.ac.uk

Phil McMinn
Department of Computer Science
The University of Sheffield
UK
p.mcminn@shef.ac.uk

***Abstract*—This paper is concerned with the challenge of reorganising a software system into modules that both obey sound design principles and are sensible to domain experts. The problem has given rise to several unsupervised automated approaches that use techniques such as clustering and Formal Concept Analysis. Although results are often partially correct, they usually require refinement to enable the developer to integrate domain knowledge. This paper presents the SUMO algorithm, an approach that is complementary to existing techniques and enables the maintainer to refine their results. The algorithm is guaranteed to eventually yield a result that is satisfactory to the maintainer, and the evaluation on a diverse range of systems shows that this occurs with a reasonably low amount of effort.**

***Keywords*-clustering; remodularization; constraint solving**

I. Introduction

The problem of software modularisation is well established. The arrangement of software entities (e.g. classes or files) may be unsuitable either from a comprehension standpoint, or for technical reasons [1]. This has to be addressed by reorganising them into a more suitable configuration (i.e. *remodularising* them), which can be performed as part of maintenance.

Identifying a suitable modularisation is a challenging, time-consuming process. It should obey good design principles (e.g. minimal coupling and maximal cohesion)but these should also be balanced against the fact that the final modules should make sense to the developer or maintainer. For example, a pair of elements might not be particularly interdependent from a source code point of view, yet might still belong into the same module because they serve a similar functional purpose. Conversely, a pair of elements might contain numerous interdependencies in the source code, but could nonetheless belong in different modules (e.g. a unit that makes extensive use of a general utility class).

Manually remodularising a system is challenging because for any nontrivial system there is a vast number of possible module configurations. Assessing every possible configuration individually in terms of design heuristics and their suitability with respect to the domain is generally intractable. Instead, the pragmatic developer can at best refactor the system in a piecemeal fashion, using individual localised refactorings within the time constraints available.

Several researchers have attempted to address the problem by developing fully automated remodularisation approaches. These tend to employ 'unsupervised' Machine Learning techniques such as clustering and Formal Concept Analysis (FCA). Both approaches ultimately use certain indicators in the system to assign elements into appropriate groups. Previous approaches have selected indicators to be file names [2], the connectivity of the files [3] or other features [4], [5], [6], [7], [8]. They are unsupervised in the sense that there is no (or at best very limited) means to externally guide the process to improve the final outcome.

The rationale for this work is captured by an observation by Glorie *e*t al. [9]. In their work on applying the most popular software clustering approach (Bunch [3]) alongside an FCA-based approach in an industrial context (Philips Medical Systems), they observed that these unsupervised clustering approaches tended to produce results that were *"'non-acceptable' for the domain experts"*. The key finding was that, whilst several modules were (largely) sensible, many others were nonsensical. Some clusters were too large, some were too small, some simply did not make sense because they joined together elements that were completely unrelated. Furthermore, they noted that there was no practical means to refine the results (i.e. by splitting a specific clustering), whilst ensuring that other desirable features (i.e. clusterings that are known to be correct) are maintained. Often a new solution is produced that is not necessarily an improvement; although it incorporates the refinement added by the user, it can also lose lots of the features that the user would have liked to have kept.

This paper introduces a *supervised* remodularisation technique (SUMO). Starting from an initial modularisation (which may be the current modules in the software system, or a modularisation proposed by existing modularisation tools), it iteratively enables this modularisation to be refined by incorporating domain information from the software developer. This information can either be *positive* or *negative*. The developer might know for example that *"Classes* $XMLParser$ *and* $AbstractParser$ *belong together, but*

neither should be in the same module as $DataVisualizer$.". SUMO will iteratively incorporate this knowledge, formulate new (improved) modularisations that obey these constraints, and invite the developer to add further refinements, until the developer is content (i.e. cannot find any further corrections).

The contributions of this paper are as follows:

- A set-theoretical characterisation of the remodularisation search problem, which clearly illustrates its expense (Section II).
- The constraint-based supervised remodularisation algorithm (SUMO) that complements existing unsupervised approaches (Section III-B).
- An analysis of the worst-case performance of the algorithm, and proof that it will produce an ideal result in polynomial time (Section III-C).
- An evaluation with respect to a diverse set of software systems, indicating that the approach tends to require a reasonably low amount of input to converge at a result that is ideal (to the developer) (Section IV).

II. Background

This section discusses the search space of the software (re-)modularisation problem. The purpose is twofold; firstly, it illustrates the sheer scale of the remodularisation challenge and the difficulty faced by automated and semi-automated techniques. Secondly, it presents a simple lattice-based representation of the search space. Besides illustrating the scale of the problem, it also provides a rationale for the approach presented in this paper, along with several basic definitions that will be used in Section III to describe the SUMO technique.

The problem of software modularisation can be thought of in set-theoretical terms. Given a set of software modules M, the task is to identify suitable groupings of these modules. Such a set of groupings is referred to as a *set partition*.

Definition 1. *A* partition *of a set M defines a set of n groups $\{S_1, \ldots, S_n\}$ such that $\bigcup\{S_1, \ldots, S_n\} = M$, where $1 \leq n \leq |M|$.*

For a given set of software elements, the search space consists of their possible partitions. For a set of size n the number of partitions (known as the Bell number of n) grows steeply. To illustrate, for $n = 4$ (i.e. four classes) the number of partitions is 15, for $n = 5$ the number is 52, and for $n = 6$ the number is 203. If we consider the modularisation of a moderately sized system with 200 classes, the Bell number (i.e. the number of possible partitions) is approximately $6.247 * 10^{276}$.

There is an intrinsic partial order that governs the possible partitions. Intuitively, a solution that includes all of the elements in one big module is more general (or *coarser*) than a solution that divides the same elements into subgroups. This *partition refinement* relation forms a *partition lattice* over the possible solutions, and is defined as follows.

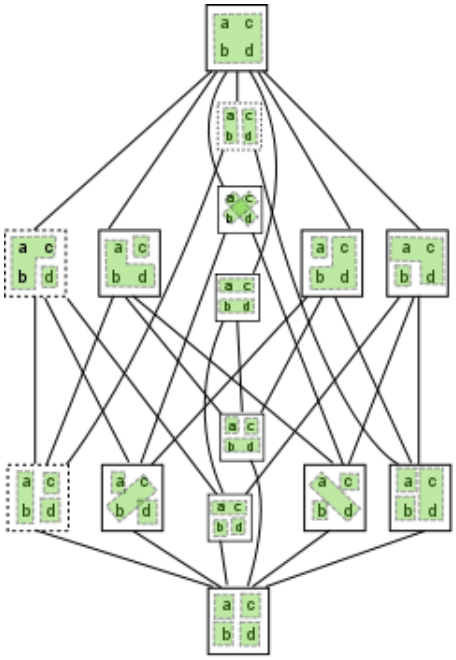

Figure 1. Example of a partition lattice for four elements (a, b, c, and d). Each node represents a different partitioning.

Definition 2. *In a* partition lattice*, the most general element $\top$ denotes the partition where all of the elements belong to a single set, and the most specific element $\bot$ denotes the partition where each element belongs in its own set. The rest of the partitions are ordered according to the* partition refinement *relation $\preceq$. For any pair of partitions P_1 and P_2, $P_1 \preceq P2$ if for every set $a \in P_1$, there exists $b \in P_2$ such that $a \subseteq b$.*

Finding a suitable software modularisation can be interpreted as trying to find a suitable node in the partition lattice. For our SUMO approach however, the developer is able to supply some constraints about which pairs of elements belong together (Rel^+) and which pairs do not (Rel^-). Thus, the challenge is to find a node in the lattice that is *consistent* with these.

Definition 3. *A partition of M is* consistent *with Rel^+ and Rel^- if for every relation $(i, j) \in Rel^+$, there exists a module $m \in M$ such that $\{i, j\} \subseteq m$. Conversely, for every relation $(k, l) \in Rel^-$ there should exist no module $n \in M$, where $\{k, l\} \subseteq n$.*

Figure 1 illustrates the partition lattice for four elements (a, b, c, and d). It illustrates how, for a given relationship between a pair of elements, the number of solutions that conform to it constitute a relatively small proportion of the search space. This is the rationale for the SUMO approach. It shows that providing a relatively small amount of knowledge in the form of relations Rel^+ and Rel^- and focussing

only on solutions that are consistent with these can lead to a substantial reduction in the search space. For example, for $Rel^+ = \{(a, b)\}$ and $Rel^- = \{(a, d)\}$, only three of the fifteen possible solutions are consistent (highlighted in dashed rectangles in Figure 1).

In Machine Learning terminology, the sub-lattice of solutions that are consistent with the input data (in our case Rel^+ and Rel^-) is referred to as a *version space* [10]. Since we will draw upon this concept to define the success criteria for our approach, a more formal definition is provided below.

Definition 4. *In intuitive terms a version space $VS(Rel^+, Rel^-)$ represents the subset of solutions that conform to Rel^+ and Rel^-. It can be defined in terms of two boundary sets of solutions: the set of most general (or coarse) solutions VS_G and the set of most specific solutions VS_S. Formally, $VS_G = \{p|(\forall q \in VS(Rel^+, Rel^-))q \preceq p\}$. Similarly, $VS_S = \{p|(\forall q \in VS(Rel^+, Rel^-))p \preceq q\}$. The lattice of solutions in between VS_G and VS_S represents those solutions that can be obtained by merging modules in VS_S or splitting modules VS_G in such a way that remains consistent with Rel^+ and Rel^-.*

The next section presents a technique that takes advantage of the search space. It will show how these sets of relationships can be acquired from the user's domain knowledge in an interactive fashion and how this knowledge can be used to refine the results produced by tools such as Bunch.

III. Supervised Remodularisation

The SUMO (**Su**pervised Re**mo**dularisation) algorithm provides a process within which it becomes possible to iteratively feed domain knowledge into the remodularisation process. It uses constraint solving, and is guaranteed to eventually produce a result that will satisfy the user (in the sense that they will find no modules that need to be reconfigured). Although it does not rely upon clustering techniques, it can benefit from them in the sense that outputs from existing tools can be used as starting points for further refinement.

The SUMO algorithm works by presenting hypothesised modularisations to the user, who will agree with some relations, and disagree with others. The developer's corrections can be integrated into the modularisation process, in turn leading to a new modularisation, which can again be refined. This forms a 'virtuous cycle' of conjectures and refutations [11], where each new hypothesis results in further corrections, gradually aggregating the requisite domain knowledge (in the form of constraints) that is required to produce a modularisation that the developer is satisfied with.

This section provides a basic overview of the algorithm, followed by a more in-depth analysis of the constraint-solving part that identifies the hypothesis modularisations.

Input: Mod
Data: Rel^-,Rel^+,$solved$,H,$NewPos$,$NewNeg$
Uses: $solve(X, Y)$, $identifyCorrections(X)$
Result: H

```
1  Rel+ ← ∅;
2  Rel− ← ∅;
3  solved ← false;
4  H ← Mod;
5  while (¬solved) do
6  |  (NewPos, NewNeg) ← identifyCorrections(H);
7  |  if (NewPos ∪ NewNeg = ∅) then
8  |  |  solved ← true;
9  |  else
10 |  |  Rel+ ← Rel+ ∪ NewPos;
11 |  |  Rel− ← Rel− ∪ NewNeg;
12 |  |  H ← solve(Rel+, Rel−);
13 |  end
14 end
15 return H
```

Algorithm 1: SUMO algorithm

This is followed by an analysis of the worst-case performance of the algorithm, and finally with a brief overview of our proof-of-concept implementation.

A. SUMO Algorithm

The SUMO algorithm is presented in Algorithm 1. It takes as input an existing partition of the system Mod – this might be the current directory structure of the system, or a modularisation proposed by existing tools such as Bunch [3]. It uses the two (initially empty) sets Rel^+ and Rel^- to store pairs of elements (m_i, m_j) that respectively should or should not belong together. The process essentially consists of a loop that presents a hypothesis partition H to the user (the $identifyCorrections$ function). If they can identify no corrections the process terminates. If corrections are identified, they are added to Rel^+ and Rel^- (depending on whether they are positive or negative). The $solve$ function then produces a new partition that is consistent with these augmented constraints. The $identifyCorrections$ function is briefly described below. The $solve$ function is given a more detailed treatment in Section III-B.

The $identifyCorrections$ function requires little elaboration. The current modularisation is presented to the user, who has the option of selecting any relations that they agree with, and any that are clearly incorrect. One accessible approach would be to present the modularisation in a graphical format, so that the user could visually inspect and select individual relations or entire modules and tag them as correct or incorrect. The function should also enable positive or negative relations that were selected in previous iterations to be highlighted as such, to prevent the mistaken introduction of inconsistent relations.

B. Software Modularisation as a Constraint Satisfaction Problem

The $solve$ function in the SUMO algorithm has the challenge of producing a partition of software modules that is consistent with the relations in Rel^+ and Rel^-. This is essentially a constraint satisfaction problem and can, as such, be solved by existing solvers.

Given a set of elements $E = \{e_0, \ldots, e_n\}$, their possible modules can be represented as a number $\mathbb{N} = [1 : n]$. The challenge is to find a set of assignments (i.e. a partition) $p : E \rightarrow \mathbb{N}$, where each element in E is mapped to a value denoting its module. The constraints on the possible mappings of p are contained within Rel^+ and Rel^-. Rel^+ is a set of pairs of elements, where the presence of a pair (e_i, e_j) implies that $p(e_i) = p(e_j)$. Similarly, the presence of a pair (e_i, e_j) in Rel^- implies that $p(e_i) \neq p(e_j)$.

As a trivial example, suppose three elements $E = \{XMLParser, DataVisualizer, AbstractParser\}$, with $Rel^+ = \{(XMLParser, AbstractParser)\}$
and $Rel^- = \{(XMLParser, DataVisualizer),$
$(AbstractParser, DataVisualizer)\}$ are to be clustered. In terms of a constraint program (using the above encoding) this might look as follows:

```
XMLParser = [1:3]
AbstractParser = [1:3]
DataVisualizer = [1:3]
XMLParser == AbstractParser
XMLParser != DataVisualizer
AbstractParser != DataVisualizer
```

As output, a constraint solver will either provide an assignment for the elements that is consistent with Rel^+ and Rel^-. For example, $XMLParser = 1$, $AbstractParser = 1$, $DataVisualizer = 2$. Alternatively, it will return an error message stating that the problem cannot be solved. The latter situation can arise from the fact that the developer has provided inconsistent constraints (though this can be mitigated by the $IdentifyCorrections$ function as discussed above).

As mentioned in Section II, the possible range of partitions forms a sub-lattice of the partition lattice, known as the *version space* [10]. The version space concept is useful because it provides a means to define when the iterative modularisation process converges, *i.e.* when it is no longer possible for $IdentifyCorrections$ to yield any further corrections. Every time a relation is added to Rel^+ or Rel^-, the version space contracts. Over multiple iterations possible modularisations are added to and removed from VS_S and VS_G, as they gradually move closer together in the lattice. Once $VS_S = VS_G$ (the most specific set of solutions and most general set of solutions are the same), the process has *converged* [10], in the sense that no further information can be added that will change the contents of the modules. This represents the ultimate success criterion for SUMO - where it has sufficient information to definitely define a single set of modules [1].

C. Worst Case Running Time

What is the maximum number of constraints that will need to be added to Rel^+ and Rel^- to guarantee convergence? Before providing an empirical answer to assess performance for realistic software systems, we first consider the theoretical worst case.

Theorem 1. *For a system with n elements, the number of relations that are required to ensure convergence is bounded by $\mathcal{O}(\frac{n(n-1)}{2})$.*

Informal Proof: We presume no initial knowledge about the system, so $Rel^+ = \emptyset$ and $Rel^- = \emptyset$. Furthermore, we presume that the $identifyCorrections$ function only provides negative feedback (relations stating that two elements do not belong in the same module). These are of the least value to the constraint solver, since inequalities are non-transitive and are much less effective at reducing the search space than positive relations. For every iteration, we assume that the function $solve(Rel^+, Rel^-)$ always selects an unsatisfactory solution where possible (which will necessarily require a further input by $identifyCorrections$).

Given the above setting, we choose the solution that requires the largest number of negative inputs to definitively characterise, which is most specific solution in the lattice where each element belongs to its own module. To converge in this setting, $identifyCorrections$ will thus be required to explicitly return the negative relationship between every distinct pair of elements. Characterising the number of elements in Rel^- for this scenario will therefore give us the general worst-case.

The number of relationships between different elements in a set is relatively straightforward to count. For example, the elements in Rel^- for a system with 6 elements (a,b,c,d,e,f) would be:

```
(a,b) (a,c) (a,d) (a,e) (a,f)
(b,c) (b,d) (b,e) (b,f)
(c,d) (c,e) (c,f)
(d,e) (d,f)
(e,f)
```

The above set of relations can be counted as 5+4+3+2+1 (adding up the number of relations in each row). This is the triangular number of 5, which is calculated as $5 * (5 + 1)/2 = 15$. In other words, the worst case for n elements is calculated as the triangular number of $n - 1$, which can be rewritten for n as $\frac{n(n-1)}{2}$. ■

Clearly this is a pathological scenario and, as will be shown in the Evaluation in Section IV, under more real-

[1]The version space may contain lots of solutions when it has converged, as there will be multiple solutions to a set of constraints that correspond in practice to a single modularisation.

istic assumptions the performance tends to be much more favourable. However, this limit is nonetheless interesting because it provides an insight into the value of this approach. Although there is a combinatorial explosion in the number of possible modularisations for n elements, these all arise from the configuration of a much smaller number of pairwise relationships. It is this observation that is exploited in this paper; by constraining certain interrelationships between software elements, the number of possible modularisations of these elements is reduced by several orders of magnitude.

As discussed in Section II, for n elements the number of possible partitions is the Bell number of n. Processing these modularisations individually rapidly becomes intractable. However, this is not necessarily the case when considering the sets of modularisations in terms of the relationships between elements. For a system of 200 classes, although there may be $6.247 * 10^{276}$ possible partitions, using the above limit, the number of interrelationships between classes that would need to be specified to guarantee an exact solution is bounded by the (relatively much smaller number of) $\frac{200*199}{2} = 19900$ interrelations between the classes.

D. Implementation

To evaluate the approach, a small proof-of-concept implementation has been constructed in Java. Initial module configurations are supplied in the form of textual SIL files – produced by Bunch that define the clusters and their members. These may either be provided by tools such as Bunch, or be simply encodings of the current software modules. The $solve$ function is implemented with the Choco constraint solving library [12].

IV. Evaluation

This evaluation seeks to assess the performance of SUMO in its intended usage context – starting from an initial module configuration and refining this to match the developer's intuitive preferences. The performance of SUMO is measured in terms of (1) the number of iterations required to converge, and (2) the quality of the proposed configuration at each iteration. The feedback provided by the user is simulated, and the target is a synthesised variant of the existing modularisation of the system.

The research questions this evaluation answers are thus:

- RQ1 How much effort is required to produce a satisfactory modularisation from an initial starting point generated by Bunch?
- RQ2 What is the relationship between the effort and the improvement obtained?
- RQ3 How does the approach scale with respect to the size of the software system?

A. Method

Figure 2 summarises the experiment methodology. The basic process is to synthesise a starting modularisation and a target (using LimSim [13] and Bunch [3] respectively). The target modularisation is used to identify random constraints between elements (simulating the human user), which are fed to SUMO over a number of iterations until the process converges and no further corrections can be identified. The detailed steps are elaborated below.

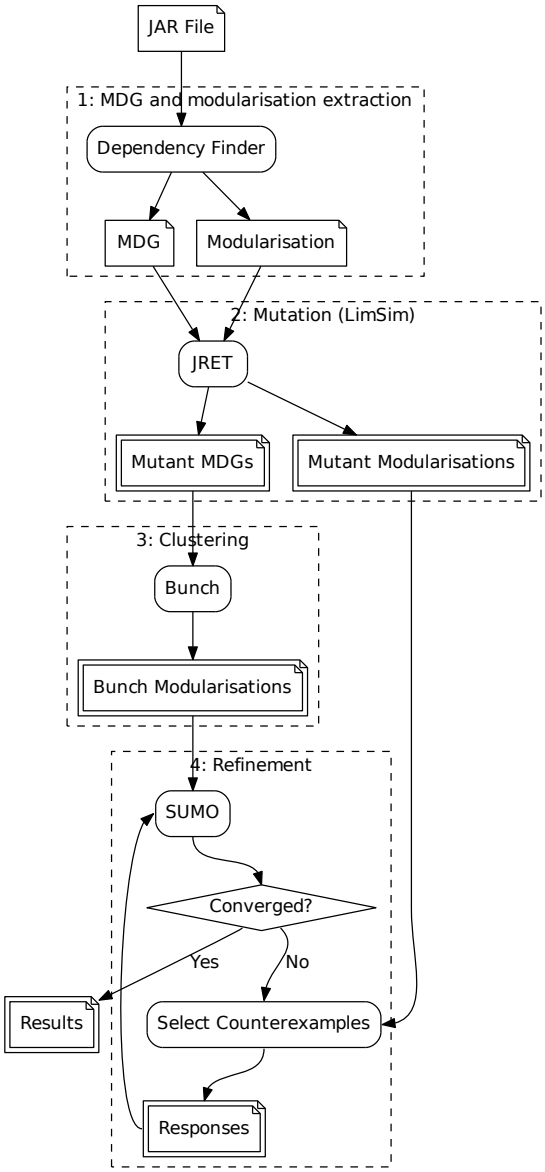

Figure 2. Data flow for the experiments; rounded edges indicate processes and the sheet icon indicates data - doubled icons signify plurality.

1) MDG and Modularisation Extraction: The purpose of this step is to produce a reference modularisation and a dependency graph for Bunch. The reference modularisation represents an ideal target that the software maintainer would be most satisfied with and is generated from the package structure of the case study. The module dependency graph serves as input to Bunch, which produces the starting point

for the SUMO algorithm.

2) Mutation: In order to ensure statistical validity, and to avoid the possible bias introduced by using one authoritative modularisation, multiple MDGs and corresponding modularisations are required. 30 pairs of MDG and corresponding reference modularisation are produced by adopting the method espoused by the LimSim clustering evaluator [13], which mutates the original MDG and modularisation. This produces 30 representative variants (the figure of 30 was chosen to mitigate sampling bias).

3) Clustering: Each variant MDG is clustered using Bunch [3]. Because this process involves a high degree of stochasticity due to the use of a hill climber, this is also repeated 30 times, again to reduce bias. This results in a total of 900 starting points for each case study. Bunch was chosen because of its popularity and its extensive use in empirical studies on clustering benchmarks [14], [13].

4) Refinement with SUMO: The purpose of this step is to apply the SUMO algorithm, and in the process to simulate the responses that would be provided by the human user (i.e. the $identifyCorrections$ function). To enable the automated application to the large base of 900 subject systems, the responses have to be simulated. Accordingly, relations between elements that either confirm or contradict a given hypothesis are selected at random from the reference modularisation produced in step 2. The number of inputs for a given iteration is selected at random from a Poisson distribution with a mean of 5. This mean was selected on the premise that, given any clustering, it would seem reasonable for a developer to be able to pick out 5 correct/incorrect relations. The long tail of the Poisson distribution accounts for situations where the clustering is either so correct, or so incorrect, that it is trivial to identify large numbers of correct/incorrect relations.

B. Measurement

At each SUMO iteration step, the MoJo difference [15] between a solution from SUMO and the mutant reference modularisation is recorded. MoJo calculates the difference between two modularisations for a system in terms of the number of modifications that must be made to one to transform it into the other.

The effort required by the developer is measured in the number of iterations SUMO requires. In other words, the number of user-interaction steps it took before SUMO produced a solution. It is important to note that (as discussed above) each interaction step elicits multiple units of feedback from the user. Although the number of individual units of feedback were recorded as well, since these were strongly correlated with the number of interaction steps, the results are presented purely with respect to the latter.

C. Subject Systems

The experiments were conducted on 5 diverse open source Java software systems. Table I summarises the case studies used in the experiment. The SLoC metric is taken as the current project size as reported by Ohloh, an open source project metrics tracker[2]. The "Component Used" column identifies which part of the system was used in cases where the product comprises multiple JAR files. Each case study was run through the procedure described in Section IV-A.

D. Results

1) RQ1: Effort Required for Convergence: Table II summarises the number of iterations required for convergence. It shows that the mean number of iterations taken to converge varied significantly between each case study, ranging from 19.34 at the lowest to 223.76 at the highest. As will be discussed with respect to scalability (RQ3) there are several factors that play a role, especially the number of modules in the target configuration, and the quality of the initial set of modules (as produced by Bunch).

2) RQ2: Improvement Versus Effort: To provide an intuition of the increase in accuracy for each iteration, Figure 3 shows 10 randomly selected runs of the SUMO algorithm for each case study. A sample is used to reduce the visual noise. Each line tracks the quality improvement at each iteration step. The quality metric is calculated at each step by taking a solution from SUMO and calculating the MoJo distance between it and the reference modularisation. This is then mapped to a percentage using the formula $percentage = 100(currentMoJo - startMoJo)/startMoJo$.

The lines firstly illustrate that the difference in quality between iterations does not necessarily increase monotonically. Although there is a cumulative improvement, it is often possible for the quality to decrease from one step to the next. This is in part due to chance on the part of the constraint solver; it might happen to pick a solution that, despite fulfilling the additional constraints, happens to be worse overall than the previous solution. However, over multiple iterations the quality will tend to increase, simply because the accumulated constraints will gradually reduce the search space to such an extent that such errors become less of a possibility.

For two of the case studies (zxing and wiquery), there is an apparent point of dimishing returns. Beyond a certain point further individual iterations contribute relatively little to the quality of the hypothesis. For the zxing example this is particularly apparent. Within the first 100 iterations there is a quality improvement of around 80-90%, but it takes a further 70-100 iterations to eventually converge.

Finally, there is a degree of variance in performance between different runs of SUMO, a trend that applies to a greater or lesser extent to all of the case studies. For example, for the collections case study, the best performing trajectory converges at just over 60 iterations, and increases much more steeply from the start than the others. On the

[2]http://ohloh.net/

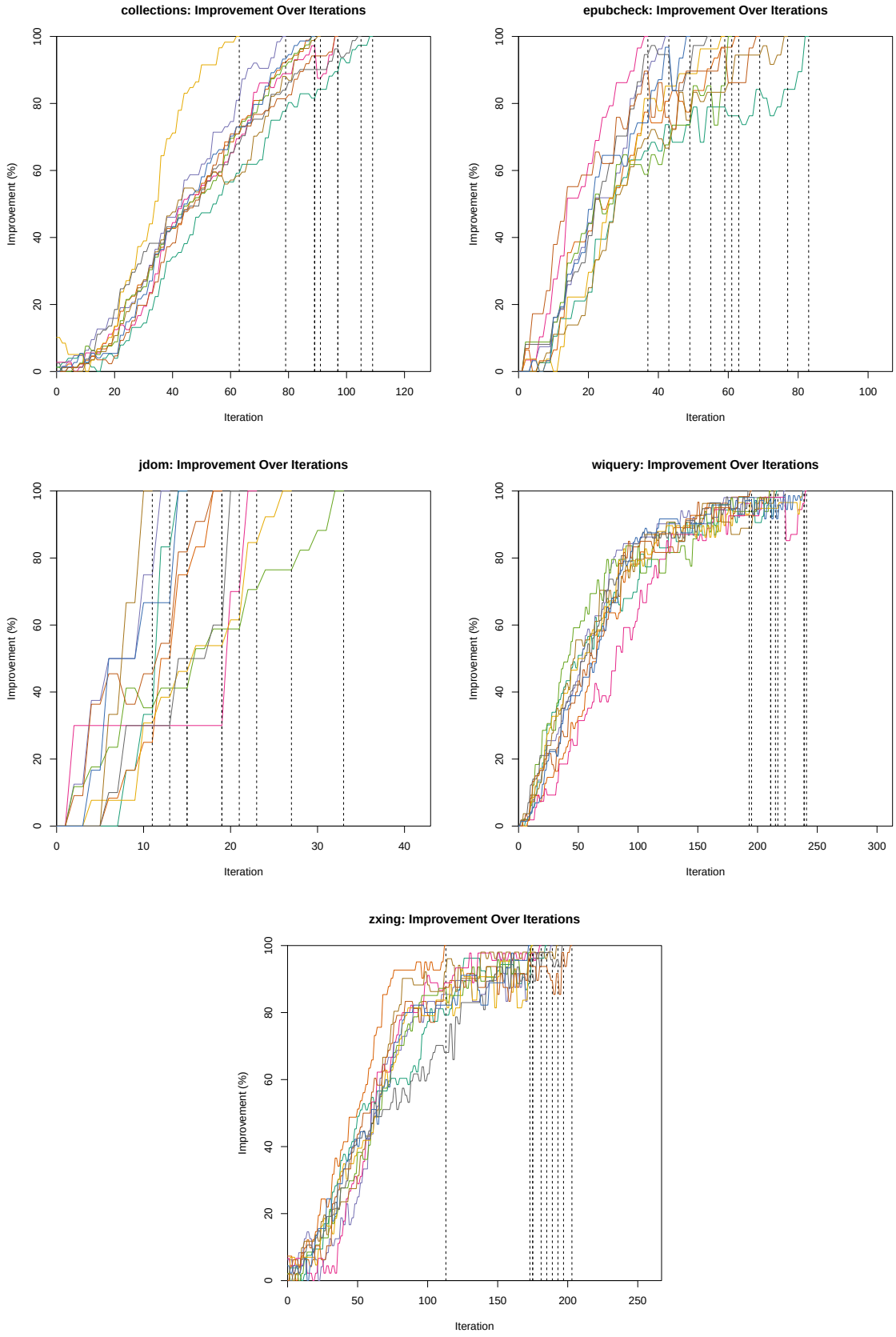

Figure 3. Percentage improvement for a random sample of 10 SUMO runs for each case study; vertical dashed lines indicate convergence for each run.

Name	Description	Version	Size (SLoC)	Component Used
Collections	Apache Commons Collections library, consisting of classes that implement various data structures	3.2.1	151,727	`commons-collections-3.2.1.jar`
epubcheck	Java tool that validates ebooks in the epub file format.	3.0-b4	17,477	`epubcheck-3.0b4.jar`
JDOM	Java XML parsing library	1.0	10,972	`jdom-1.0.jar`
Wiquery	Library to integrate the jQuery JavaScript library with the Wicket Java web framework	1.5-M1	79,267	`wiquery-1.5-M1.jar`
ZXing	Java barcode scanning library.	2.1-SNAPSHOT	161,063	`core.jar`

Table I
CASE STUDIES USED IN THE EXPERIMENTS

Case Study	Mutant MDG Size:				Reference modularisaton:	Start MoJo	Iterations to Percentile:							
							25%		50%		75%		100%	
	Min	Max	Mean	St.Dev	N^o Modules	Mean	Mean	St.Dev	Mean	St.Dev	Mean	St.Dev	Mean	St.Dev
collections	453	523	492.57	14.96	12	73.38	29.12	4.93	43.99	7.17	62.93	8.92	86.47	12.58
epubcheck	138	191	159.13	12.92	13	35.43	16.10	4.06	27.16	6.44	42.99	10.52	62.61	13.13
jdom	124	183	146.57	12.68	7	9.91	8.04	3.44	12.48	4.35	16.62	5.80	19.34	6.48
wiquery	314	401	365.63	17.16	29	60.69	35.45	8.05	64.72	11.32	101.82	19.79	223.76	26.25
zxing	262	327	294.30	18.04	37	44.92	37.66	10.40	59.35	12.78	83.93	15.88	179.99	25.68

Table II
SIZES OF THE MUTANT MDGS GENERATED FROM EACH STUDY AND THE NUMBER OF STEPS REQUIRED TO REACH EACH PERCENTILE OF CLOSENESS TO THE TARGET MDG

other hand, in the wiquery example, the worst performing trajectory fluctuates more extensively, and consistently under-performs the others. In general, trajectories that start off well tend to carry on performing well and converge earlier, whereas trajectories that start off poorly tend to take longer to converge, and don't improve as rapidly.

This clearly indicates that the choice of feedback to SUMO plays an important role. Different relations provided by the user can be much more capable of shaping and restricting the search space than others. For example, if the user states in the first iteration that five elements belong together, this would be a much stronger and more valuable statement than the fact that they do not belong with five other elements. As shown in the results, providing useful feedback early on means that subsequent solutions are more accurate, and enable the user to provide more informative feedback about other aspects of the partitioning, which has the cumulative effect of leading to a stronger overall improvement.

3) RQ3: Scalability: There are two key factors that affect the scalability of SUMO: the number of clusters in the target modularisation, and the quality of the starting modularisation.

The mean number of clusters in the target modularisation is presented in Table II. It shows that zxing and wiquery have a larger number of target modules than the others. The increased number of iterations for these systems is reflected in the scatter plot in Figure 4. Irrespective of the quality of the clustering score, these two systems require a much larger number of iterations to converge than the others. Indeed, the Pearson correlation between the number of clusters in the target and the number of iterations required is $r = 0.906$.

However, the plot also makes it evident that there is a

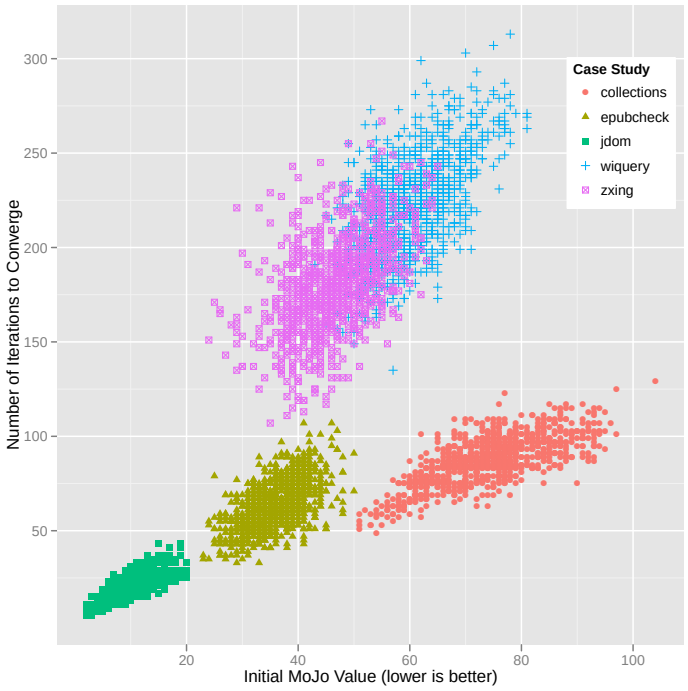

Figure 4. Scatterplot of the number of iterations taken and the initial quality of each modularisation

further factor at play. The clusters tend to adopt a shape that is skewed diagonally from the lower left to the upper right. This indicates a clear relationship between the starting quality of the modularisation (i.e. the Bunch result) and the number of iterations required. This is intuitive; a more accurate initial modularisation simply means that fewer

corrections need to be factored in by SUMO.

E. Threats to Validity

There are several factors in the experiment that constitute threats to validity. The key factors are discussed below.

The constraint solver could bias the results: Given a set of constraints, there is usually more than one possible solution in the version space. The choice of solution is clearly significant, as this forms the basis for the provision of subsequent constraints. It is possible that certain strategies that a constraint solver might adopt (e.g. to try to always provide the most general solution) would in turn bias the selection of constraints by the user. It is possible that more suitable constraint solvers might exist, or that better heuristics could be adopted to select solutions – investigating these is a part of our ongoing work.

The use of Bunch to produce a starting point could bias the results: The Bunch tool is used to generate the starting points for SUMO. Its search based approach means that there is an inherent variance in the quality of its output. To allay the risk that this will affect the validity of the results, Bunch and SUMO are executed 30 times on the same MDG.

The automated responses might not accurately represent those chosen by a human user: The simulated user responses could be unrealistic in terms of (1) the volume of constraints, (2) the choice of specific constraints, and (3) the ratio of positive to negative constraints. The number of constraints are chosen according to the Poisson distribution at a mean of 5, specific constraints are chosen in a quasi-random fashion, and the ratio of positive to negative constraints is not controlled at all. There is a strong argument that, ultimately, a human user with the ability to resort to their domain knowledge and intuition is likely to make much better choices, and to select much larger numbers of constraints if necessary. This is a factor that we are currently investigating in more extensive experimental studies.

The subject systems, the synthesised starting MDGs, and target modularisations may not be representative of general software systems: The evaluation uses just five software systems from which to derive its conclusions. This sample is too small to argue that it is representative – that the performance profile of SUMO on these systems would be similar in any other software system. Furthermore, these systems are not used as-is, but are used as input to the LimSim methodology, which uses mutation to generate a larger population of starting systems and target modularisations. Given the random nature of the mutants (they are selected and applied at random), there is the additional danger that the result ends up being unrealistic. For example, applying a large number of mutations to a particular MDG might cumulatively dilute what was an initially well-defined modular structure, thus artificially making the outcome harder to modularise.

It is important to bear this threat in mind when interpreting the results; these cannot be used to draw reliable conclusions about the performance of SUMO with respect to general software systems. However, this was not the point; the research questions assess facets of the performance of SUMO that apply to any system; the relations between number of iterations required, the quality of the initial modularisation and the intricacy of the target modularisation. In this respect, the selection of the case study systems, coupled with LimSim, provides a comprehensive range of subject systems that are reasonably diverse, and which provide us with statistically justified insights. The question of how SUMO performs in a more realistic context, with respect to a larger range of software systems, is being addressed in ongoing work.

V. Related Work

Supervised software remodularisation: Modularisation as "software renovation" has been previously applied with a semi-interactive step that allows the user to re-run a search-based algorithm until it produces a satisfactory result [16]. This differs from the interactive approach applied by SUMO in that the user's positive or negative responses are only used to rerun the whole algorithm, which gradually accumulate a body of information about the underlying system as is the case in this work.

Bunch is equipped with methods designed to improve the quality of the clustering, including detection of libraries and omnipresent files; these serve a purpose of cutting down the search space and focusing it only on the modules that are likely to be relevent. "User driven" clustering is also available in Bunch, however this is non-interactive and only captures positive examples. For example, it is only possible for the user to express that a and b should be together, but not that a and c should not be together. Efforts to improve on the performance of Bunch have mainly been focused on the search technique itself [17], [18] in order to maximise fitness.

Constraint acquisition: The process used in this paper to build a set of constraints is related to the work by Bessiere *et al.* on query-driven constraint acquisition [19]. Their CONACQ approach addresses a similar problem, in the sense that they are attempting to reason with a constraint solver about an incomplete system of constraints, and seek further input from a human user. However, whereas they formulate specific constraints as queries to the user, the approach used here presents the user with a full hypothesis solution (the modularisation of the system), and asks them for counter-examples. In Machine Learning terminology, whereas they pose 'membership-queries' (requiring a simple 'yes' or 'no' as an answer), the approach proposed here uses 'equivalence queries' [20], requiring the user to provide at least one counter-example. There would certainly be scope for the use of membership queries in this work (i.e.

do modules x and y belong together?). However, in the software maintenance usage scenario, where the maintainer does not necessarily have the patience to answer what could be thousands of questions, it seems more practical to rely on their intuition to select those counter examples that seem most obvious.

VI. Conclusions and Future Work

This paper has presented a semi-automated framework that is guaranteed to produce modularisations that are satisfactory to the user. It shows how existing modularisations (whether extant in the software or proposed by other tools) can be refined in a systematic fashion.

The evaluation on a small but diverse selection of real software packages indicates that there is a consistent increase in the quality of the results as more input from the developer is provided. The principal factor affecting the effort required from the user is the number of modules in the target output; a more intricate target requires more input. However this is also affected by the quality of the starting point. If the initial modularisation is of a low quality, more effort will be required again. However, in any case the use of SUMO (even for a small number of iterations) is likely to lead to significant improvements in the final modularisation. This clearly addresses weaknesses in existing unsupervised techniques, as described by Glorie *et al.* [9], where there is a desire to refine modularisations by feeding-in domain knowledge.

Future work will expand the evaluation to a larger set of case studies, and involve human subjects to provide constraints. This will follow the development of the current proof-of-concept implementation into an interactive tool.

Although the current $identifyCorrections$ function will reject a constraint that is inconsistent with the existing set of constraints, this makes the possibly questionable assumption that the user is always consistent. Future work will attempt instead to incorporate their confidence, to allow such conflicts to be resolved or tolerated with the help of fuzzy logic.

Acknowledgments. This work is supported by the EPSRC grant REGI: EP/F065825/1.

References

[1] G. Antoniol, M. Di Penta, and M. Neteler, "Moving to smaller libraries via clustering and genetic algorithms," in *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings*, 2003, pp. 307–316.

[2] N. Anquetil and T. Lethbridge, "Recovering Software Architecture from the Names of Source Files," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 3, pp. 201–221, 1999.

[3] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 50–59.

[4] N. Anquetil, C. Fourrier, and T. C. Lethbridge, "Experiments with hierarchical clustering algorithms as software remodularization methods," *Working Conference on Reverse Engineering 1999*, 1999.

[5] A. van Deursen and T. Kuipers, "Identifying Objects using Cluster and Concept Analysis," in *ICSE '99 Proceedings of the 21st international conference on Software engineering*. ACM, 1999, pp. 264–255.

[6] P. Tonella, "Concept analysis for module restructuring," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, Apr. 2001.

[7] R. Lutz, "Recovering High-Level Structure of Software Systems Using a Minimum Description Length Principle," *Artificial Intelligence and Cognitive Science*, pp. 63–80, 2002.

[8] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *Software Engineering, IEEE Transactions on*, vol. 31, no. 2, pp. 150–165, 2005.

[9] M. Glorie, A. Zaidman, A. Van Deursen, and L. Hofland, "Splitting a large software repository for easing future software evolution - an industrial experience report," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 113–141, 2009.

[10] T. M. Mitchell, "Generalization as search," *Artificial Intelligence*, vol. 18, pp. 203–226, 1982.

[11] K. R. Popper, *The logic of scientific discovery*. Hutchinson:London, 1959.

[12] N. Jussien, G. Rochart, and X. Lorca, "The CHOCO constraint programming solver," in *CPAIOR08 workshop on OpenSource Software for Integer and Constraint Programming OSSICP08*, 2008.

[13] M. Shtern and V. Tzerpos, "Evaluating software clustering using multiple simulated authoritative decompositions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 353–361.

[14] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *2005 IEEE International Conference on Software Maintenance (ICSM)*, 2005, pp. 525–535.

[15] V. Tzerpos and R. C. Holt, "MoJo: a distance metric for software clusterings," *Proceedings of the 6th Working Conference on Reverse Engineering*, pp. 187–193, 199.

[16] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, "A language-independent software renovation framework," *Journal of Systems and Software*, vol. 77, no. 3, pp. 225–240, 2005.

[17] K. Praditwong, M. Harman, and X. Yao, "Software Module Clustering as a Multi-Objective Search Problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.

[18] K. Mahdavi, M. Harman, and R. Hierons, "A Multiple Hill Climbing Approach to Software Module Clustering," in *ICSM '03 Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2003, pp. 315–324.

[19] C. Bessiere, R. Coletta, B. O'Sullivan, M. Paulin *et al.*, "Query-driven constraint acquisition," in *Proc. Twentieth Internatioal Joint Conference on Artificial Intelligence-IJCAI*, vol. 7, 2007, pp. 50–55.

[20] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.