Effectively Incorporating Expert Knowledge in Automated Software Remodularisation

Mathew Hall, Neil Walkinshaw, Phil McMinn

Abstract—Remodularising the components of a software system is challenging: sound design principles (e.g., coupling and cohesion) need to be balanced against developer intuition of which entities conceptually belong together. Despite this, automated approaches to remodularisation tend to ignore domain knowledge, leading to results that can be nonsensical to developers. Nevertheless, suppling such knowledge is a potentially burdensome task to perform manually. A lot information may need to be specified, particularly for large systems. Addressing these concerns, we propose the SUMO (SUpervised reMOdularisation) approach. SUMO is a technique that aims to leverage a small subset of domain knowledge about a system to produce a remodularisation that will be acceptable to a developer. With SUMO, developers refine a modularisation by iteratively supplying corrections. These corrections constrain the type of remodularisation eventually required, enabling SUMO to dramatically reduce the solution space. This in turn reduces the amount of feedback the developer needs to supply. We perform a comprehensive systematic evaluation using 100 real world subject systems. Our results show that SUMO guarantees convergence on a target remodularisation with a tractable amount of user interaction.

Index Terms—software remodularisation, domain knowledge, set partitioning

---

1 INTRODUCTION

REMODULARISATION remains a difficult and unsolved problem in software maintenance. As software evolves to meet new requirements, its design invariably deteriorates, making it harder to maintain. To moderate this deterioration, systems can be *remodularised* so that their components are configured in a way that enables, for example, comprehension [1] or performance [2]. The task of restructuring a system by hand tends to be prohibitively time consuming and resource intensive [3].

Existing remodularisation algorithms have sought to produce improved designs automatically, but have been unable to do so satisfactorily [3], [4]. Automated algorithms have focused on using techniques such as clustering or formal concept analysis [5], [6], [7], [8], [9], [10], [11] to produce groupings for software components. However, these techniques formulate modules by focussing on the source code alone, and consequently tend to produce solutions that do not make sense from a conceptual point of view [3], [4], [12]. For example, an industrial study [3] of the popular Bunch remodularisation tool [11] found that Bunch's results were *"non-acceptable for the domain experts"* for the task of reorganising software components for a large medical system, consisting of several million lines of code.

These limitations have led to the realisation that software remodularisation techniques must necessarily involve a degree of input from an expert. Accordingly, several variants of existing modularisation algorithms have been developed, which seek to accommodate this need [11], [13]. However, they tend to be limited in practical terms as they either (a) interrogate the user for feedback in a way that renders them prohibitively expensive, or (b) fail to provide guidance to the user, leaving them with no indication of how much input is necessary or of value to the underlying algorithm.

In order to address these problems, we introduce the SUMO (SUpervised reMOdularisation) technique [14]. SUMO is based on the observation that existing general purpose clustering algorithms can be improved with relatively little domain knowledge [15]. Remodularisation algorithms often produce *partial solutions* [3], [4], but given a set of corrections, these partial solutions may be transformed into desired modularisations. For example, given a proposed clustering for a data processing framework, a developer might make an observation that contradicts the current proposed solution, such as *"Classes XMLParser and AbstractParser belong together, but neither should be in the same module as DataVisualizer."* SUMO provides a mechanism by which to enable the developer to feed-in this corrective information in the form of specific relationships, for example *"XMLParser does not belong with DataVisualizer"*.

SUMO treats the problem of software remodularisation as a set partitioning problem, where the elements of the set are the entities of the system, and the partitions are the modules. SUMO then converts corrective feedback from the user into constraints on the possible set partitions, which can be solved using a constraint solver (a tool that searches for assignments for a set of variables subject to constraints on their values). This not only guarantees that the user's preferences are respected, but also significantly limits the amount of input required on the part of the user. Unlike other approaches [16], SUMO does not require some notionally "complete" set of inputs to produce a result, but will continuously maintain a "hypothesis" that can be shown to the user, and which is adapted as new corrections are provided.

We conduct an analysis of the theoretical limits on corrective feedback that SUMO may require, and examine its

- *M. Hall and P. McMinn are with the Department of Computer Science, University of Sheffield, UK. N. Walkinshaw is with the Department of Computer Science, University of Leicester, UK. E-mail: mathew.hall@sheffield.ac.uk. This research was funded by the EPSRC Grant EP/F065825/1.*

performance compared to these bounds on a benchmark of 100 subject systems. We find that in all applicable cases (92), SUMO is guaranteed to converge on a target modularisation using a tractable amount of domain knowledge.

This work builds upon some preliminary research [14] that investigated the feasibility of the approach. The early version of the approach did however suffer from the significant limitation that it was very difficult to use in an iterative way; every time a user supplied some feedback, they could potentially be faced with a completely new proposed set of modules. This lack of continuity rendered the approach very difficult to use in practice. There was also no proposition of how the proposed modularisations should be presented to a user, and only a limited empirical study.

In this paper we present a revised version of SUMO that explicitly supports continuity between iterations by incorporating previous solutions where possible. We have implemented a graphical interface to shows how inputs can be solicited from an end-user. We have also substantially extended our empirical analysis to account for a much larger range of subject systems.

The contributions of this paper are therefore as follows:

1) The SUpervised reMOdularisation (SUMO) algorithm, which views the remodularisation problem as one of set partitioning (Section 3). SUMO refines a modularisation by soliciting feedback iteratively from a domain expert. This feedback is formulated as a series of constraints over a set partition, which are solved by a constraint solver (Sections 4.1–4.2).

2) A theoretical analysis of the SUMO algorithm, detailing best and worst case limits on the quantity of information required to guarantee convergence (Section 4.3).

3) An empirical evaluation of SUMO with respect to a diverse set of software systems, indicating that in practice the approach tends to require a reasonably low amount of input to converge to a suitable remodularisation. (Section 5).

The rest of the paper begins with further background to the problem of automatic software remodularisation, and the need to incorporate domain knowledge. In Section 3 we characterise software remodularisation as a set partitioning problem, showing how the difficulty of the problem can be reduced given the availability of additional constraints. Section 4 then introduces the SUMO algorithm, which leverages set partitioning and additional user-specified constraints to converge on desirable modularisations. We give theoretical bounds on SUMO's performance, and describe its implementation into a tool. In Section 5 we describe an empirical evaluation of SUMO on 100 real world software systems, drawn from SourceForge. We then discuss the SUMO approach in terms of existing work in Section 6. Following this we conclude the paper in Section 7 with possibilities for further avenues of research.

2 BACKGROUND

In this section, we briefly discuss the rationale for software remodularisation. We then present an overview of the existing research into automating the approach, covering techniques that are partially or fully automated. For both families of approaches we provide some of the key barriers that have prevented them from being adopted in practice.

2.1 Software Remodularisation

Software systems are conventionally modular [1]; core elements (classes or other components) tend to be grouped together according to their common functionality or purpose within the system. As software evolves, elements within the system are repurposed to address changes in requirements and the core purpose of different modules can become diluted, making the system harder to understand as a whole.

Remodularisation is the challenge of reversing this deterioration by reorganising the software to improve its modular structure. The number of possible re-organisations for a system is huge, and developing a solution can require an intractable amount of effort and knowledge from the developer.

2.2 Automated and Semi-Automated Solutions

Numerous automated techniques have been developed that seek to minimize the manual effort involved in remodularisation. These can be broadly categorised into techniques that are fully automated and require no user intervention (i.e., unsupervised techniques), and techniques that incorporate a degree of input from a user.

2.2.1 Automated Approaches

Unsupervised techniques operate by expressing the remodularisation task as a generic clustering problem. Functions are produced that can measure the relative "distance" between two classes, for example in terms of the number of interdependencies, or the similarities of their identifiers. Once these are in place, clustering algorithms can be applied to propose new clusters (i.e., modules).

Over the past 30 years, numerous such clustering-based approaches have been proposed [5], [6], [7], [9], [10], [11], [17], [18]. One of the most prominent approaches is the Bunch tool [11], which uses a genetic algorithm to derive clusters that optimise the MQ measure [11] — a function that measures the ratio of inter-module and intra-module dependencies (i.e., coupling and cohesion).

Unfortunately, current automated approaches fail to produce satisfactory results [4]; although a grouping of elements might make sense from a design perspective (e.g., being heavily interdependent), it might not make any sense from the perspective of the domain model [3], [19].

2.2.2 Semi-Automated Approaches

To attenuate the inaccuracy of fully automated approaches, some techniques have been adapted to allow for a degree of manual intervention. The amount of input required can vary substantially from technique to technique. Some approaches are intrinsically reliant upon the user; they compute a "domain" of possible modularisations, but rely on the user to choose the final set of modules. Others are more lightweight; they will start from a proposed set of modules, but rely on the user to provide appropriate corrections.

As an example of the former (more heavyweight) family of techniques, several approaches that are based upon

Formal Concept Analysis (FCA) have been proposed [7], [8]. Using this approach, the user identifies again a set of features (e.g., identifiers or type-usages), and every file or class to be modularised is categorised according to these attributes. As a result, FCA computes a "concept lattice", where each node in the lattice represents a potential module. The user is then required to select the nodes to yield the final remodularisation.

There are numerous examples of approaches that permit a more light-weight form of feedback from the user. Users might be permitted to tune the parameters of the algorithm [10], [11], [13], [20], [21], [22]. Other approaches allow more fine-grained control over the algorithm's decisions [11], [13] — that is, by providing hints to constrain the range of solutions that it can produce.

Rigi [20] provides an editor for a modularisation that presents an example of the parameter-tuning approach. It allows the user to manually reorganise the system, but also provides component identification through the application of graph-based heuristics. The user can select which approach to apply and the appropriate weights. Bunch [11] also allows the user to identify components that may be problematic for the optimisation algorithm such as libraries, and commonly used components.

Bunch [11] and the more recent Interactive Genetic Algorithm (IGA) [16] are examples of techniques that enables more direct user interaction. Both enable the user to interact directly with the underlying genetic algorithm as it performs the clustering. Similarly, Arch [13] allows the user to confirm or reject merges suggested by a hierarchical clustering algorithm.

Glorie et al. studied the use of Bunch and FCA-based approaches in an industrial environment [3]. They found that the approaches they studied were unable to appropriately accommodate the feedback provided by users, and that the approaches were too effort-consuming to be practical.

One of their key findings was that the tools they chose were difficult to use because they were difficult to control. The process of converging on a final modularisation was not gradual and refinement-driven. Every iteration, the algorithms would propose entirely new solutions that were incorrect in unanticipated ways. They made two recommendations for the combination of module clustering and domain knowledge:

> "When domain knowledge is available and you want to use that domain knowledge to define a rough initial clustering, make sure that your clustering tool (1) supports a pre-defined starting solution and (2) does not tear this initial clustering apart."

The problems mentioned by Glorie et al. highlight a bigger issue that still besets semi-automated remodularisation approaches. Approaches need to provide users with a familiar frame of reference against which to provide feedback (i.e., an existing, recognised modularisation) [19]. Proposed remodularisations need to accommodate user feedback, but need to retain a degree of familiarity so as not to disorientate the user. Perhaps most importantly, they need to offer some form of guarantee that the technique will eventually converge on a solution that suits the user.

Addressing these concerns, we present SUMO, which allows users to iteratively refine a modularisation of a system. SUMO recharacterises the problem of remodularistation as a set partitioning problem, which we introduce in the next section.

3 SOFTWARE REMODULARISATION AS A SET PARTITION PROBLEM

The purpose of recasting the problem of software remodularisation into one of set partitioning is twofold: Firstly it illustrates the 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 space of possible remodularisations, which provides the means to incorporate developer preferences in the remodularisation process, and forms the basis of the SUMO approach presented in this paper.

Given a set of software entities E (which may be files, classes, or other entities [3]), the objective of a software remodularisation process is to identify a suitable grouping of these entities into modular components. This is equivalent to the set partitioning problem, where a set of such groupings is referred to as a *partition*:

Definition 1 (Partition). *A **partition** of some set E involves a set $\mathbf{M}$ of components (modules) $\{M_1, \ldots, M_m\}$ such that $E = \bigcup\{M_1, \ldots, M_m\}$, and $\bigcap\{M_1, \ldots, M_m\} = \emptyset$, where $1 \leq m \leq |E|$.*

For a set of software elements, the search space includes all possible partitions. The number of possible partitions for a set of size n is given by the Bell number [23] of n. This number grows extremely steeply, making even small partitioning problems intractable; for example, for $n = 4$, there are 15 possible partitions. This number grows to 52 for $n = 5$, and 203 for $n = 6$. One of the tasks in Glorie et al.'s industrial case study [3] involves the remodularisation of 109 entities. Accordingly, there are 1.096×10^{129} possible partitions. As a system grows, so does the number of possible configurations of its components, compounding the difficulty in remodularising it by hand.

Our approach exploits the fact that the search space of the possible partitions (solutions) is governed by an intrinsic partial order. A solution that includes all elements in one module is more general (or *coarser*) than a solution that divides the elements into subgroups. This *partition refinement* relation forms a *partition lattice* over the possible solutions.

Definition 2 (Partition Lattice). *In a **partition lattice**, $\top$ denotes the most general partition where all of the elements belong to a single set, and $\bot$ the most specific partition where each element belongs in its own set. The remaining 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 identifying a suitable node in the partition lattice. The developer will be aware of some constraints (informed by domain knowledge of the elements, or opinion about the system architecture) that ought to hold between pairs of elements; that is, certain elements should belong in the same module, or should be kept apart. This knowledge is
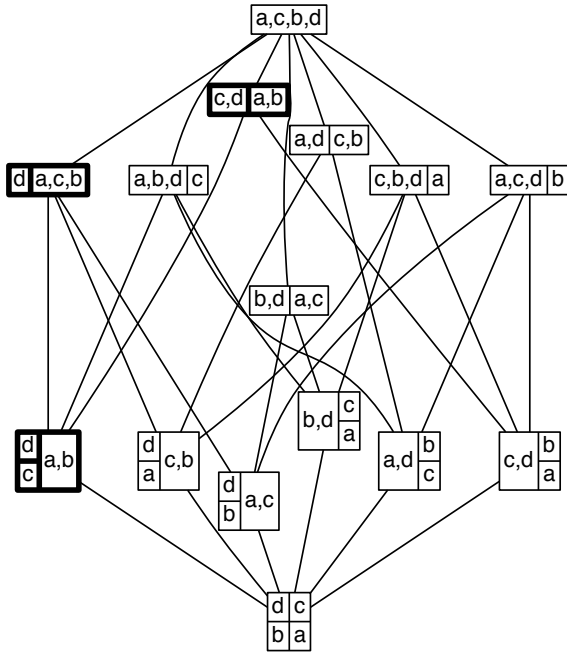
Fig. 1. Example of a partition lattice for four elements (a, b, c, and d). Each node represents a different partitioning.

held in two sets, Rel^+ and Rel^-, respectively. The challenge is to find a node in the lattice (i.e., a modularisation) that is *consistent* with the developer's conceptual view of the system (i.e., the information in Rel^+ and Rel^-).

Definition 3 (Consistency). *The developer's preferences are denoted as the sets Rel^+ and Rel^- (the sets of relations indicating that elements belong together or apart respectively). A partition of M is **consistent** with Rel^+ and Rel^- if for every relation $\{i, j\} \in Rel^+$, there exists a module $M \in \mathbf{M}$ such that $\{i, j\} \subseteq M$. Conversely, for every relation $\{k, l\} \in Rel^-$ there should exist no module $M \in \mathbf{M}$, where $\{k, l\} \subseteq M$.*

The set-theoretical illustration of the software modularisation problem enables us to cast the search for a suitable modularisation as a traversal of the partition lattice of possible solutions. Although the theoretical size of this lattice is huge, the number of solutions within the lattice that are consistent with Rel^+ and Rel^- can amount to a relatively small fraction.

Figure 1 illustrates the partition lattice for four elements. On the face of it, for just four elements there are 15 possible solutions. It shows how the number of solutions that conforms to a given relationship constitute a relatively small proportion of the space of possible modularisations. For the example in Figure 1 and the two relations $Rel^+ = \{\{a, b\}\}$ and $Rel^- = \{\{a, d\}\}$, only three of the fifteen possible solutions are consistent (highlighted with bold borders).

This property – that the supply of a relatively small amount of external information can substantially reduce the search space – forms the basis of our SUMO approach, which will be presented in full in Section 4. Naturally, Figure 1 is merely a conceptual illustration of the technique. The number and nature of constraints required, and the extent to which the search space is reduced, will be revisited in full in the evaluation.

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* [24].

Definition 4 (Version Space). *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^-.*

By re-characterising the search for a suitable modularisation in these terms, we can identify the "ideal" modularisation (consistent with Rel^+ and Rel^-) if the version space contains only one solution. In other words, the two boundary sets VS_G and VS_S are equal and contain one element.

The question of what constitutes an "ideal" modularisation is ultimately subjective; different users could have differing opinions on which elements do or do not belong together. The next section introduces the SUMO algorithm, which allows a user to iteratively feed their opinions and domain knowledge (Rel^+ and Rel^-) into the remodularisation process. Using a constraint solver to produce a solution $M \in VS(Rel^+, Rel^-)$, SUMO is guaranteed to be consistent with the user's corrective feedback and eventually converge on the user's ideal solution. Although it does not use clustering techniques itself, SUMO can use outputs from existing tools, such as Bunch [11], as a starting point for refinement.

4 THE SUMO APPROACH

The SUMO approach 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 [25], 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. This is followed by an analysis of the worst-case performance of the algorithm, and a brief overview of our implementation of the SUMO algorithm into a tool with a GUI.

4.1 SUMO Algorithm

Figure 2 summarises the steps of the SUMO algorithm, which we elaborate in more detail in Algorithm 1. The algorithm relies on two types of information that influences the solutions it generates:
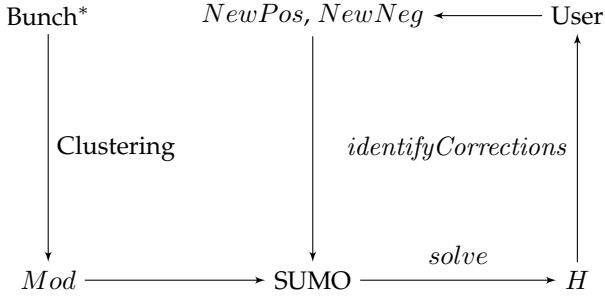
1) An initial modularisation, *Mod*

Fig. 2. The SUMO process: starting with an initial modularisation Mod, SUMO repeatedly presents the hypothesised solution H and refines it according to relations supplied by the user. In this example Bunch (highlighted by the asterisk) generates the starting point; in practice other remodularisation algorithms can be used.

ALGORITHM 1: SUMO algorithm

> **Input**: Mod
> **Data**: Rel^-,Rel^+,$solved$,H,$NewPos$,$NewNeg$
> **Uses**: $solve(X, Y, Z)$, $identifyCorrections(X)$
> **Result**: H
> $Rel^+ \leftarrow \emptyset$;
> $Rel^- \leftarrow \emptyset$;
> $solved \leftarrow false$;
> $H \leftarrow Mod$;
> **while** $(\neg solved)$ **do**
> > $(NewPos, NewNeg) \leftarrow identifyCorrections(H)$;
> > **if** $(NewPos \cup NewNeg = \emptyset)$ **then**
> > > $solved \leftarrow true$;
> > **else**
> > > $Rel^+ \leftarrow Rel^+ \cup NewPos$;
> > > $Rel^- \leftarrow Rel^- \cup NewNeg$;
> > > $H \leftarrow solve(Rel^+, Rel^-, H)$;
> > **end**
> **end**
> **return** H

---

2) Corrections Rel^+ and Rel^-

The algorithm begins with an initial modularisation Mod, which may be the current package structure, or a proposed modularisation from a tool such as Bunch [11]. The algorithm builds on this solution by iteratively soliciting feedback from a user and applying a constraint solver to produce new solutions.

The first most significant part of the main loop is the call to the $identifyCorrections$ function, which presents a hypothesised modularisation H to the user, and allows them to supply the *positive* (two elements should be together in a module) and *negative* (two elements shouldn't be together) relations belonging to the sets Rel^+ and Rel^- respectively. For instance, these corrections could be to better align the modularisation with their domain knowledge, or for architectural or design reasons (e.g., to place two classes together that play a similar role in a design pattern).

Figure 3 shows how such relations can be elicited from the user by presenting the hypothesised solution through a wireframe GUI that allows the user to add in positive and negative information. Figure 3a shows a wireframe in which the hypothesised modularisation H is shown to the user. Figure 3b shows how the user may add relations by creating edges between elements in the user interface,

(a) Initial set of modules

(b) Following developer-supplied domain knowledge

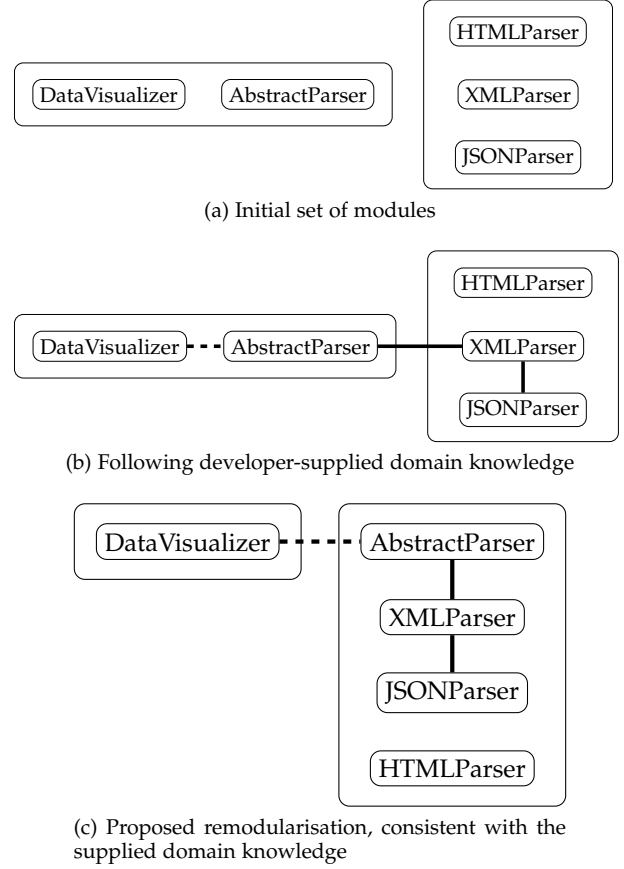(c) Proposed remodularisation, consistent with the supplied domain knowledge

Fig. 3. Eliciting relations from the user with SUMO. Solid lines indicate positive information, dashed lines indicate negative relations

which are added to the Rel^+ and Rel^- sets (solid lines indicate positive information, dashed lines indicate negative relations). Figure 3c then shows a solution revised to take account of this feedback, presented in the next iteration of the loop.

Following feedback supplied by the user through the $identifyCorrections$ function, the $solve$ function locates a partition that is consistent with the new set of constraints, retaining as much of the previous solution H (or input modularisation Mod) as possible. We describe how the $solve$ function computes a partition for a given set of constraints in the next section.

4.2 Constraint Solving in SUMO

Producing a partition of elements that is consistent with the relations in Rel^+ and Rel^- is a constraint satisfaction problem that existing solvers can provide solutions for. Given a set of elements $E = \{e_0, \ldots, e_n\}$ in a system, we represent each distinct module to which they potentially belong as a unique number drawn from the set $\mathbb{N} = [1 : n]$. The solver must then find a set of assignments (i.e., a partition) $p : E \to \mathbb{N}$, where each element in E is mapped to a number that denotes its module. The constraints on the possible mappings of p are contained within Rel^+ and Rel^-.

A pair $\{e_i, e_j\}$ in Rel^+ implies $p(e_i) = p(e_j)$. Similarly, the presence of a pair $\{e_i, e_j\}$ in Rel^- implies that $p(e_i) \neq p(e_j)$. Figure 4 shows an example in which the elements of the set E are to be remodularised subject to the given

Elements:

$E = \{XMLParser, JSONParser$
$\quad HTMLParser, DataVisualizer$
$\quad AbstractParser\}$

Variables:

```
XMLParser      = [1:5]
JSONParser     = [1:5]
HTMLParser     = [1:5]
AbstractParser = [1:5]
DataVisualizer = [1:5]
```

Relations:

$Rel^+ = \{\{XMLParser, AbstractParser\},$
$\quad\quad\quad \{XMLParser, JSONParser\}\}$
$Rel^- = \{\{AbstractParser, DataVisualizer\}\}$

Constraints:

```
XMLParser      == AbstractParser
XMLParser      == JSONParser
AbstractParser != DataVisualizer
```

Fig. 4. Representing remodularisation as a constraint program. Elements (left) become variables, and user-supplied relations (right) are translated into constraints.

relations in Rel^+ and Rel^-. The lower portion of the figure shows how these can be translated into a constraint program that can be solved by existing constraint solvers.

The constraint solver will produce an assignment for the elements that is consistent with Rel^+ and Rel^-. For the example, the assignments $XMLParser = 1$, $JSONParser = 1$, $HTMLParser = 1$, $AbstractParser = 1$, $DataVisualizer = 2$ satisfy the constraints, and is the solution shown visually by Figure 3c.

The user may supply constraints that are contradictory. If this occurs, the solver will fail, and *identifyCorrections* will indicate the problem to the user who must then remove the contradicting relations before the SUMO loop can continue.

A key challenge is to enable SUMO to *efficiently* converge on the correct result — that is, to minimize the number of elements in Rel^+ and Rel^- required from the developer. If the constraint solver picks particularly poor solutions that require a lot of corrections from the developer, SUMO will be too burdensome to use. As observed by Glorie et al. [3], large segments of the modularisations produced by automated tools such as Bunch can be "correct" according to the developer. This implies that corrections should be relatively localised.

Accordingly, SUMO biases the constraint solver to focus on solutions that are as close as possible to previous solutions. This takes place in the *solve* function. The search for a solution by the constraint solver consists of changing assignments until consistency is achieved with the relationship set. Accordingly, to keep new solutions similar to old solutions, we set the initial assignments to those of the previous solution H (or Mod).

Using the example in Figure 3a, the solver would be first initialised such that `DataVisualizer` and `AbstractParser` were set to, for example the value 2 indicating they are in the same package. Accordingly, the remaining variables would be assigned to another value, 1

in this example. After the feedback is provided, the solver will search for changes to these assignments resulting in a consistent solution. In this example, as `AbstractParser` is linked to `XMLParser`, the value of `DataVisualizer` can be changed to 1 to move it without making any other assignments.

Elements of the input without user-supplied constraints will be unchanged by SUMO, and will reflect the solution as proposed by the original clustering tool used to produce the initial modularisation. Biasing the solver in this way allows SUMO to build upon the good parts of the modularisation Mod, generated by tools such as Bunch.

4.3 Theoretical Performance Bounds

In this section, we conduct an analysis of the bounds on the efficiency of the SUMO algorithm. We consider (a) the maximal number of relations that could be added to Rel^+ and Rel^-, and (b) the minimal number of relations required to guarantee that SUMO produces the ideal result.

Theorem 1 (Maximum number of relations). *For a system with n elements the number of relations that can be provided is bounded by $\frac{n(n-1)}{2}$.*

Proof. The maximum number of relations is given by the number of distinct pairs of entities (there can only be one relation for each distinct pair). This can be calculated by the triangle number of the number of entities: $\frac{n(n-1)}{2}$.

For example, suppose the five elements from the example in Section 4.2 are to be clustered. A maximum of $\frac{5*4}{2} = 10$ unique relations can be added between each distinct pair of elements:

$\{AbstractParser, DataVisualizer\}$ $\{DataVisualizer, JSONParser\}$
$\{AbstractParser, XMLParser\}$ $\{DataVisualizer, HTMLParser\}$
$\{AbstractParser, JSONParser\}$ $\{XMLParser, JSONParser\}$
$\{AbstractParser, HTMLParser\}$ $\{XMLParser, HTMLParser\}$
$\{DataVisualizer, XMLParser\}$ $\{JSONParser, HTMLParser\}$

$\square$

The above bound is pathological. The only theoretical circumstance that could require this amount of input to guarantee a correct result is if the user opts to supply a relation for all pairs of elements (and continues to do so even if SUMO produces the desired result). In reality, the developer will supply fewer relations than this maximum, so we wish to determine the minimum number of relations required to guarantee that SUMO will converge.

Theorem 2 (Minimal number of relations). *For a system containing n entities, the minimum number of positive and negative relations required to guarantee convergence on a solution with m modules is given by $|Rel^+| \geq n - m$ and $|Rel^-| \geq \frac{m(m-1)}{2}$.*

Proof. We need to quantify the minimum number of constraints between elements to guarantee that $solve(Rel^+, Rel^-)$ returns the correct result. Rel^+ and Rel^- must be sufficiently complete that any relation not in the union of these sets can be deduced (by exploiting the transitivity of positive information).

Let $G = (E, Rel^+ \cup Rel^-)$ be a graph, of the entities in the system (E) forming vertices, and the relation sets forming labeled edges in G.

Considering positive relations first, a positive edge must exist between each element within a module. Positive edges are transitive, so it is sufficient such that each module is a connected component in G. For each module $M \subseteq E$, at least $|M| - 1$ positive edges are needed to make it a connected component. Summation over all m modules gives $|Rel^+| = |E| - m$. This solution, however, only guarantees all of the elements will be grouped together, so one single module satisfies this constraint. We must add negative information to prevent this and guarantee convergence. If a negative edge is added between a pair of elements in different modules, which are connected, then it can be inferred for all other pairs of elements between the modules. Thus, only one edge is needed between each distinct pair of modules: $|Rel^-| = \frac{m(m-1)}{2}$. $\qquad\square$

These bounds give insights into the value of the SUMO 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 pair-wise relationships. SUMO exploits this observation; by constraining certain relationships between elements, the number of possible modularisations of the elements is reduced by several orders of magnitude.

Section 2 showed that for n elements the number of possible partitions is the Bell number of n and that processing these modularisations individually becomes intractable very rapidly. However, this is not necessarily the case when considering the sets of modularisations in terms of the relationships between elements. With respect to the example from Glorie et al. [3], for a system of 109 classes, there are $1.096 * 10^{129}$ possible partitions. However, using the limit derived in Theorem 1, 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 $\frac{109*108}{2} = 5,886$.

Theorem 2 shows that this number may be smaller than this upper bound if there are fewer than n modules. For example, if there are 18 target modules (the number produced by Bunch in Glorie et al.'s paper), then $|Rel^+| = 109 - 18 = 91$ and $|Rel^-| \geq \frac{18*17}{2} = 153$.

4.4 The SUMO Tool

Finally, we implemented the SUMO algorithm into an open source tool which is available for download[1], so that we could observe real users interacting with it for the purposes of informing our empirical study (as described in Section 5.1). Written in Java, it uses the CHOCO constraint solver [26] to identify possible modularisations from the user's constraints. It elicits constraints from the user through a graphical interface, enabling them to link elements with positive or negative relations and lock modules when they have been correctly identified in the hypothesis solution.

SUMO takes as initial input a text file that contains all of the entities that are to be modularised (usually source code files), along with a preliminary modularisation. This initial modularisation might simply constitute their current

___

1. Available at https://bitbucket.org/mathew_hall/sumo/ and https://bitbucket.org/mathew_hall/sumo-api/

TABLE 1
Overview of the user testing sessions.

	Sheffield	Leicester
Participants	22	35
Subject	JDOM	Guava
Task Size (Classes)	57	122

organisation into directories, or it may be the output from an unsupervised clustering tool, such as Bunch [11].

The tool presents modules as visual "blobs", each of a different colour, where elements that are thought to belong together share the same blob. A screenshot of an interaction with SUMO is shown in Figure 5. The visualisation, built on the Prefuse [27] visualisation framework, is dynamic; the user can interact by dragging modules around, and zooming in and out. The zooming functionality becomes critical to enable the navigation of larger systems [22].

The user interface implements the *identifyCorrections* component of the SUMO algorithm. As described in Section 4.1, the user provides relations by creating edges between nodes in the visualisation. The visualisation reacts to these relations, positive edges cause elements to be drawn closer and negative edges cause elements to repel one another. After the user has added their corrections, they can click on a button labelled "next", which invokes the *solve* function to produce a new, refined modularisation. The tool then presents this new modularisation to the user, allowing them to supply further relations if necessary.

If a user is completely satisfied that a module is correct, it can be locked. Internally, we implemented this by adding positive relations between all of the elements in the module, and negative relations between the elements in the module and all other external elements. This enables the user to add large amounts of information in a single click.

Given that users will make mistakes (for instance, a user might add edges that are negative when they should be positive, or attempt to add relations that contradict each other), the tool provides an undo button that the user may click to remove the relations added to the Rel^+ and Rel^- sets in the last round of corrective feedback (i.e., the last invocation of *identifyCorrections*).

4.5 User Testing the SUMO Tool

During the development of SUMO, we observed how users interacted with it to identify improvements and understand what type of feedback people might provide.

We conducted two sessions with users from the Universities of Sheffield and Leicester. In each session, users received a short tutorial on remodularisation and SUMO before being invited to use SUMO on an example system. After this example, during which participants could ask questions, we directed the participants to use SUMO to remodularise a larger system, starting from a configuration generated automatically using Bunch. Table 1 summarises the two sessions we ran.

We used the Sheffield session as a pilot study and collected qualitative feedback by observing users and noting the problems they found with the tool. In response to their feedback, we implemented the solver initialisation described in Section 4.2 to alleviate the problem of SUMO
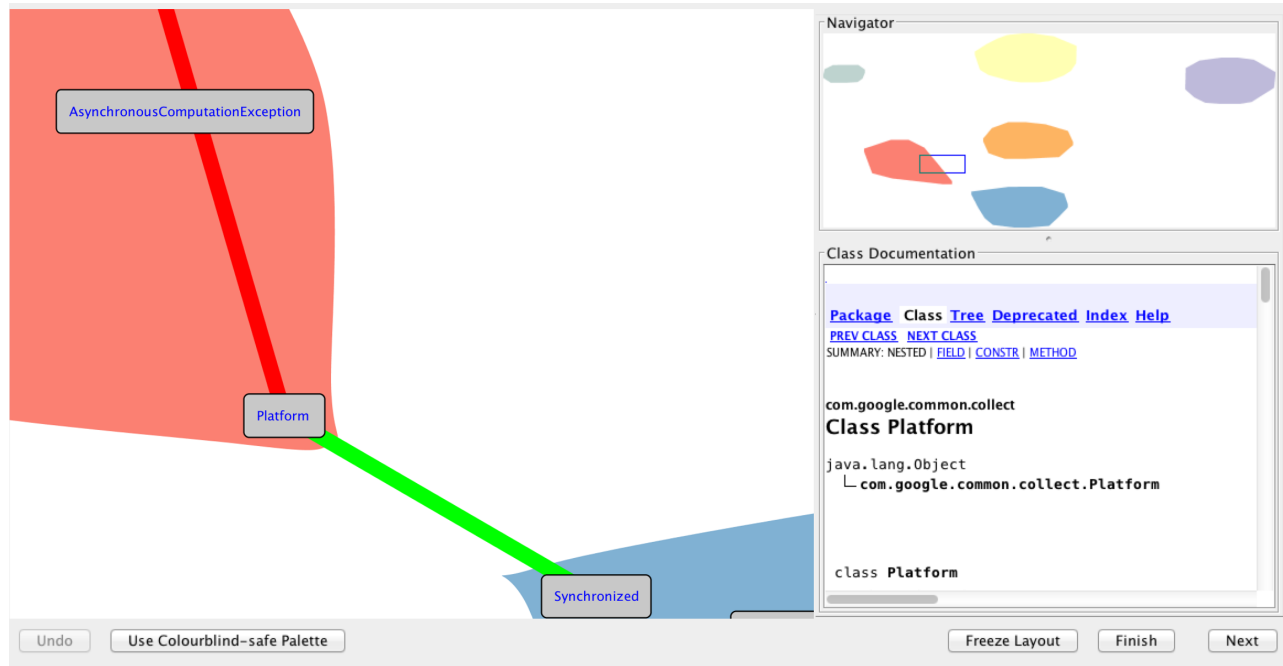
Fig. 5. The SUMO user interface, showing an overview of a hypothesised modularisation in the top right hand corner. The main panel zooms into a smaller part of this modularisation, allowing the user to add positive and negative information. SUMO aids the user by with the help of JavaDocs in the bottom right panel. In this example, the main panel shows a positive link between "AsynchronousComputationException" and "Platform", and a negative link between "Synchronized" and "Platform".

generating highly dissimilar solutions between iterations. Without any other guidance, CHOCO returns the most general solution, leading users to request new solutions less frequently and thus foregoing the advantage SUMO offers.

We used the revised SUMO algorithm we present in this paper for the user feedback session at Leicester. Encouraged by the rate at which participants of the Sheffield study completed the task, we opted to use a larger subject. Alongside the revised algorithm, we instrumented the tool to collect the relations supplied and time each request for a new solution was made. The data we collected can be used to paint an aggregated picture of how users interact with the constraint solver using the SUMO tool.

Users of the instrumented SUMO tool provided an average of 82% positive relationships, excluding those using the lock feature. On average, a user provided 4.89 relationships before requesting the next solution from the constraint solver. The median time taken to provide information was one minute and two seconds.

Alongside these aggregated measures, we observed users throughout the session and found that it was common for participants to provide non-confirmatory feedback. For example, if a relation $a + b$ and $b - c$ were provided, users may also provide the matching $a - c$ relation that is already deducible from the previous two relationships. We postulate that providing relationships serves one of two purposes for our users: identifying incorrect assignments to directly correct them; or to indicate agreement with the proposal and mark them as "done", serving as a reminder of which parts of the system they have already visited.

Our observation of users also shows a high degree of subjectivity, measuring the difference between the starting modularisation and the final one produced by each indi-

vidual (using MoJo [28]) yielded a distribution showing a high degree of discordance. The median distance between the starting point and the final modularisation was 44. The spread of this distribution, with a minimum of 3 and maximum of 73 highlights the subjective nature of the remodularisation problem.

5 EMPIRICAL EVALUATION

Although SUMO will always eventually produce the desired set of modules, its practicality is dependent upon the amount of effort (in terms of time spent providing input) that is required from the user. This, however, can depend on a multitude of factors. Considerations have to include, for example, the size of the system being remodularised, the number of modules that the user envisages in that system, and the type of input they provide (e.g. whether they focus on highlighting classes that belong together, or instead focus on highlighting 'negative' relations between classes that should remain in separate modules). This section provides an empirical evaluation of SUMO in these terms. Our specific research questions are as follows:

RQ1 How expensive is SUMO in terms of effort required from the user?

RQ2 What factors affect SUMO's performance?

5.1 Methodology

Given the multitude of (potential) factors at play, we use an automated evaluation method, running multiple configurations of SUMO on 100 different Java systems. In order to validate our findings in RQ1, we relate them to data obtained from the user testing sessions in 4.5.

5.1.1 Setup

The automated evaluation method is based on previous work by the authors [14] and Bavota et al. [16]. This approach uses a simulated user to interact with SUMO on a multitude of different subject systems, from a variety of starting configurations, and enables us to control for a variety of factors. Ultimately this allowed us to examine a larger range of systems and gather more data than an observational study would permit: our results are made up of a total of 1,380,000 individual runs of SUMO.

As starting configurations we used modularisations of the system that were suggested by Bunch [11]. This approximates the process of the industrial study described by Glorie et al. [3] wherein a user, simulated in our case, is asked to provide subjective feedback on a solution produced using Bunch. As Bunch is a randomised algorithm, we repeated the process 30 times. This generated 30 starting points for each subject system. We then developed a simple model of a user to supply SUMO with inputs on a selection of 100 open-source Java systems (both the user model and the subject systems are described below). Figure 6 outlines the process taken to acquire the results used in our analysis.

5.1.1.1 User Model: Our user model is deliberately simple: Given a target modularisation (which is taken to be the original modularisation of the subject system), it quasi-randomly selects pairs of classes, using the target modularisation to express whether or not the relation between that pair should be positive or negative. The choice is quasi-random because the user model (a) makes sure that it does not supply the same relation twice, (b) is parameterised according to the proportion of class pairs that should be positive, and (c) supplies a random number of feedback-pairs at each iteration, where this random number is governed by sampling from the distribution of feedback sizes provided by users in the user testing described in Section 4.5.

As we measure the information required to transform an input modularisation to a target modularisation, we require a suitable target that represents the developer's domain knowledge about the system. Remodularisation is a highly subjective problem, as shown in the summary of our user testing in Section 4.5. In our experiments the original modularisation is the target, which ensures the topology of the target is realistic in terms of the grouping and sizes of components.

It is important to note that, although the model determines the number of relations and the proportion of positive versus negative relations, the specific choices of relations remain random. This means that the model can easily choose relations that are of little or no utility to the remodularisation task at hand. One would expect a real developer to make more considered choices — choosing relations to either corroborate or contradict specific features. As a consequence, any results that are produced by our user model ought to be interpreted in this light — a discerning developer would probably require fewer inputs to yield a useful result than our model.

5.1.1.2 Subject Systems: For the subject systems we used the "SF100" set of Java programs [29], a set of 100 programs randomly selected from SourceForge. We chose to use a random sample of subjects to avoid the risk of bias arising

from curating a set of projects. For each of these programs, we extracted its module dependency graph (MDG) using the DependencyFinder[2] tool. Table 2 summarises each of the subjects, showing its name and size, given in terms of the number of classes in its dependency graph and packages. The systems range from one very small system with just a single class, to several systems with hundreds of classes, and one single large system with $6,817$ classes.

The input format required by the Bunch remodularisation tool is unable to represent classes without dependencies, and so they are not accounted for by the figures in Table 2. Consequently, we could not remodularise the "greencow" system as it only included one class. As such, it had an empty dependency graph, and so we removed it from our analysis. Of the remaining 99 case studies, we found one to be unusable because it exceeded one week of runtime in Bunch and could not be used.

5.1.2 RQ1: How expensive is SUMO in terms of effort required from the user?

We investigate RQ1 in terms of the time taken, in iterations, by the simulated user to remodularise each subject system with SUMO. From starting modularisations produced by Bunch, we ran SUMO using the original package structure as the target modularisation for the user model. At each iteration of SUMO, we compared the current hypothesised modularisation to the target and if there were any differences, we provided SUMO with more inputs from the user model. The process stopped when SUMO reproduced the original decomposition for the subject system.

Having observed from the user-study (Section 4.5 that approximately 75% of relations selected were positive relations (indicating that a pair of classes belonged together), we fixed this as a parameter for the user model in this RQ too (different balances are explored in RQ2). As the user model selects its inputs quasi-randomly, we ran 100 repetitions of the process for each of the 30 Bunch results for each subject system. We limited each SUMO run to 4GB of memory and 6 hours of runtime.

To study how the modularisation proposed by SUMO changed in response to feedback throughout, we measured the similarity of the hypothesised modularisation to the target at each step, and used this similarity measure as an estimate of the qualitative improvement SUMO had generated since beginning the process. To this end, we adopted the "MoJo" difference [28]. 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. It does so by estimating the number of operations to move elements between, or join clusters required. The MoJo value is an integer that is bounded by the number of entities (classes) in the data set. When the MoJo value is 0 the two modularisations are identical.

5.1.3 RQ2: What factors affect SUMO's performance?

In addition to the MoJo history and steps taken, we recorded the total number of positive and negative relations supplied by the user model to reach the target result from the

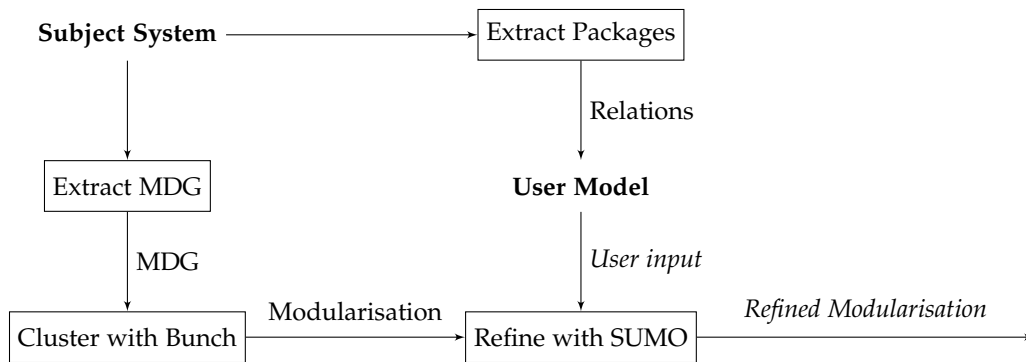---

2. http://depfind.sourceforge.net

Fig. 6. Methodology of the systematic study. We study the italicized items as outputs of the experiment. We vary the items indicated in bold.

TABLE 2
The SF100 subject systems used in the systematic study, with the number classes and packages involved in their MDG (Module Dependency Graph), as clustered by Bunch. Subject systems that exceeded the runtime or memory limits during refinement with SUMO and were excluded, and appear in italics. The system that Bunch failed to remodularise is denoted by † and the system that contained only one class is highlighted with an * (this system appears in the table as having zero classes, as the MDG only considers classes with dependencies).

MDG	Classes	Packages	MDG	Classes	Packages
a4j	45	6	*jcvi-javacommon*	981	82
apbsmem	19	1	jdbacl-0.6.9	261	25
asphodel	24	5	jgaap	27	1
at-robots2-j	296	12	jhandballmoves	109	9
battlecry	15	1	jigen	73	5
beanbin	122	23	jiggler	178	17
biblestudy	24	3	jipa	5	1
biff	6	1	jiprof	6	1
bpmail	14	5	JMCA	54	4
byuic	9	1	jnfe	190	18
caloriecount	975	112	jni_inchi	24	1
cards24	26	3	jopenchart	55	8
celwars2009	32	1	jsecurity	134	33
classviewer	24	2	jshop	34	1
corina	1005	30	jtailgui	52	10
dash	68	19	jvc	17	2
dbe	92	13	jwbf-1.3.4	98	21
dcparseargs	6	2	lagoon	90	5
diebierse	22	4	lavalamp	54	8
diffi	9	5	*lilith*	831	64
dom4j	209	14	lotus	59	10
dsachat	34	10	macaw	187	12
dvd-homevideo	36	1	mygrid	39	3
echodep	138	6	nekomud-20110901	10	4
ext4j	18	3	newzgrabber	41	1
falselight	9	1	noen	18	3
feudalismgame	59	3	nutzenportfolio	83	28
fim1	127	7	objectexplorer	84	8
fixsuite	42	5	omjstate	21	3
follow-1.7.4	96	9	openhre-hl7	111	9
fps370	7	1	*openjms*	791	44
fspath-1.0-alpha	19	3	petsoar	79	15
gae-app-manager	8	2	quickserver	107	15
gangup	51	4	resources4j	19	4
geo-google	24	4	rif	20	7
gfarcegestionfa	59	6	saxpath	16	4
*greencow**	0	0	sbmlreader2	103	6
gsftp	34	5	schemaspy-5.0.0	119	6
heal	234	37	sfm	11	4
hft-bomberman	138	17	shp2kml	6	1
httpanalyzer	77	1	*summa*	736	94
ifxf-v3.0.0-gen†	6817	2	templatedetails	24	1
imsmart	22	9	templateit-1.0-beta4	22	2
inspirento	48	6	trans-locator	6	1
io-project	16	3	tullibee-api-9.63.00-SNAPSHOT	21	1
ipcalculator	86	1	twfbplayer	143	11
javabb	44	10	water-simulator	114	9
javathena	38	11	wheelwebtool	137	10
jaw	124	10	xbus	206	42
jclo	4	1	xisemele	69	3

same Bunch starting points used for the previous research questions. We then performed an exploratory analysis to establish how various factors could affect performance. For the analysis we followed the guidelines set out by Kitchenham et al. [30] and set out the variables of interest. Our dependent variables are the steps taken and the number of positive and negative relations provided. The independent variables in our analysis are:

- The size of the system (classes)
- The size of the system (packages in target decomposition)
- The user model
- The quality of the Bunch results (MoJo distance to the target).

We related the response variables to the dependent variables using descriptive statistics and visualisation. We performed statistical tests where effects were not apparent in the descriptive statistics or visualisation. All of our analysis, including computing statistical tests, was conducted using R [31].

To establish the impact of different MoJo values on performance, we tested the hypothesis that different MoJo values will not produce differences in the number of relations required by SUMO, for solutions produced by Bunch. We performed the following analysis on each subject system for which Bunch generated more than one distinct result (initial MoJo value).

We used a Kruskal-Wallis test [32] for each of these systems, which estimates the likelihood that samples from different groups are from the same distribution. We used the Kruskal-Wallis test as it is non-parametric; values in some groups were not normally distributed (as we determined by visualising the histograms). The outcome of each test is a p-value. We set a significance level of 0.05 and controlled for multiple comparisons [33] by adjusting p-values for multiple tests using Benjamini and Hochberg's method [34].

5.1.4 Threats to Validity

The simulated user may not represent real user behaviour. The user makes a random selection of relations from the set of known-good relations irrespective of the current solution, until SUMO reproduces it completely. Given that the selection of relations is indiscriminate, there is the inherent problem that a large (possibly overwhelming) proportion of the selected relations will be spurious or irrelevant. This is the reason (discussed above) that the six largest systems yielded too many constraints. Nevertheless, the validity of the results is only threatened insofar as the results will *over-approximate* the amount of input required from a typical user — an educated software engineer would be more selective with inputs, and would likely achieve better results with less effort. Our user model can therefore be viewed as a "worst case user". We later discuss the implications of this user model in light of our user testing sessions in Section 5.2.3. Given the randomised nature of relation selection, we repeated each SUMO refinement process 100 times.

Simulating the user also means that we cannot capture the thought process that leads to constraints. By not modelling the decision process we avoid the risk that the user model outperforms a human by always providing corrective

feedback [14]. Our experiments do, however, assume that a hypothetical user would eventually be able to make judgements for a substantial proportion of the system. There is a risk that a real user may not be able to individually perform this task alone. However, SUMO does not preclude multiple users entering feedback for parts of a system for which they are responsible.

As we only have samples of the feedback generated by real users for one subject system, there is a risk that this distribution might not be identical for other subject systems. In turn, the number of steps the user model takes may not be reflective of use by a human. However, the number of relations provided (mean of 4.89) relative to the size of the subject (122 classes) in the user study suggests an overwhelming preference to provide feedback to small fractions of the system at each step, irrespective of the size of the subject.

The use of the existing package structure as a final target in the evaluation. While a real user would never choose to remodularise a software system into its existing structure, existing module structure can be used as a measure of remodularisation algorithm authoritativeness [4], and has previously been used as a proxy where no other authoritative decomposition is available [14], [16]. While there is a risk that the modular structure of the SF100 case studies are not authoritative (i.e., their developers would not create them), our results still enable us to quantify how SUMO can tailor a solution to user preferences. The impact of this choice is that we may be measuring SUMO's performance at recovering a modularisation that developers would not choose themselves. However, it still represents the customisation of Bunch-generated results to a user-specified form using an authoritative source of feedback.

The use of Bunch to produce starting modularisations. Bunch is a randomised algorithm so it is unlikely to reproduce the same solution. We mitigate the internal threat to validity that the Bunch results are atypical by repeating the clustering process 30 times per subject system (which, as previously stated, were refined by SUMO 100 times each).

We chose Bunch as an example of a remodularisation tool that could produce modularisations for SUMO to refine. While it might not be representative of all possible modularisation tools, it is one of the most widely studied (see, for example, references [4], [16], [28], [35]) and is well-suited for the task of providing a variety of starting points for SUMO to work from.

The use of SF100. We adopted the SF100 as a benchmark to remove any threat to validity that may have arisen from our own curation of a set of "suitable" subject systems. As a representative sample of SourceForge projects [29], using the SF100 benchmark allows us to generalise our results beyond the subjects we consider in our analysis. Using such a random sample means we are unable to control the size of the systems we study ourselves. As a result, some of the subject systems may be smaller than those typically requiring corrective refactoring such as remodularisation. We opt to include all the subjects in the SF100 rather than applying a threshold to the size of the system, thereby avoiding a source of bias in our results.

5.2 Results

We now discuss the results we obtained from our experiments and the answers we obtained to our research questions.

5.2.1 RQ1: How expensive is SUMO in terms of effort required from the user?

Under the control of the user model SUMO yielded the target result for all bar six of the subject systems within the six hour limit. Figure 7 summarises the number of steps taken, ordered by the median number of steps taken to converge. The plots are separated according to scale to preserve the visibility of smaller values.

The amount of input required varied substantially from system to system (and the reasons for this are explored in RQ2). For all but nine of the systems, SUMO required less than 500 iterations on average. For 48 systems, the median number of relations required was lower than 50 (often much lower). Figure 8 depicts this relationship graphically.

The average time taken by users to supply some feedback in our user testing (Section 4.5) was one minute and two seconds. Assuming this holds for other subjects, then these figures would suggest that the majority of SF-100 systems would be processed within a couple of hours, more than half being completely remodularised within an hour (probably much less if we assume that real developers make more discerning feedback choices than our user model).

For the few particularly complex systems, the amount of feedback shoots up to >1000 iterations. Of course it is unrealistic to expect a user to use SUMO for > 10 hours (if we adopt the rule of thumb of one minute per iteration). However, for such cases it is important to remember that the results here are only counting the effort required to achieve *exactly correct* solutions.

In practice, SUMO tends to produce a reasonable approximation of the target modularisation after a relatively small number of iterations (usually 50-100), whereafter a large proportion of the inputs are spent on minor improvements. Figure 10 illustrates this with respect to the echodep subject system. As a consequence, it is not necessary to to wait until the tool has converged to obtain a potentially useful result. So even though the amount of time required to achieve *exactly correct* modularisations might be prohibitively high in some cases, a small proportion of that input will tend to yield results that are at least approximately correct, and possibly suffice for typical re-engineering purposes.

In the context of the best and worst-case functions derived in Section 4.3, all subjects converged before the theoretical maximum number of relations could be supplied. The median proportion of the worst-case figure required was 0.266 across all the included subjects. In terms of the best case, SUMO required more than the theoretical minimum number of relationships to converge for the majority of subjects, requiring a median of $2.864\times$ the theoretical lower bound. For a small number of subjects, convergence was reached before the feedback reached the lower bound which is made possible by solver initialisation. If parts of the solution are already acceptable then they are retained in the absence of feedback to the contrary (see Section 4.2).

This property also holds for the six systems that failed to yield the target result within the time-limit. In all but one

subject, the MoJo value was reduced to less than 35% of its initial value. In some cases, the improvement follows a similar pattern to the series in Figure 10, in the sense that there is an initial sharp drop in MoJo distance (increase in accuracy). In two cases (caloriecount and summa), this is followed by a small 'hump', where the accuracy plateaus or decreases momentarily. This is an artefact of the user model. In both of these cases there are a large number of packages in the target modularisations, which means that the user model must supply more negative relations; the rapid rate of improvement after the plateau occurs when all positive relations are exhausted and thus more negative relations are provided at each iteration. For these large systems, the volume of possible relationships overwhelms the proportion of useful feedback, which ultimately leads to an exhaustion of runtime or memory when the selection is entirely random.

> RQ1: For the majority of systems (of the nature represented by the SF-100 dataset), a perfect solution can be obtained in 1-2 hours. In certain cases, however, obtaining a perfect solution can take more than 10 hours. An approximation of the ideal result can be achieved within the first 50-100 iterations (depending on the system).

5.2.2 RQ2: What factors affect SUMO's performance?

Our findings for RQ1 show that the amount of input required to obtain a satisfactory result with SUMO can fluctuate, depending on the system being remodularised. This research question investigates the underlying reasons. Clearly the number of classes in the systems plays a role, but there are other factors at play. This question explores the influence of the accuracy of the initial Bunch modularisation, along with system-specific factors, and the role of the proportion of positive and negative relations.

Effect of the accuracy of the initial (Bunch) modularization. Of the 92 subjects that we did not exclude as described in Section 5.1.1.2, the initial modularisations produced by Bunch resulted in variable MoJo scores for 68. For the remaining 24 subjects, the generated Bunch solutions were similar or identical. These were smaller subjects, with a median size of 19 classes.

From the subject systems that had multiple initial MoJo values we found significant differences in the amount of input required for SUMO for 6 subjects. These systems were smaller than most of the other case studies in the systematic study, the largest of which comprises 34 classes.

The remaining 62 cases with multiple MoJo values had p-values above our significance level of 0.05. In these cases, the effect of the MoJo score on the number of relations required was not statistically significant. The majority of these systems are larger (median number of classes of 68.5) than those for which the effect was significant (median number of classes = 21.5). As a result we conclude that the initial modularisation does not affect the effort required to apply SUMO unless the system is small.

Effect of the number of classes and (target) modules of the system. Computing the Pearson correlation between these two factors and the number of iterations required confirms that both have an impact. The correlation for the number of classes is 0.856, and 0.686 for the number of modules.
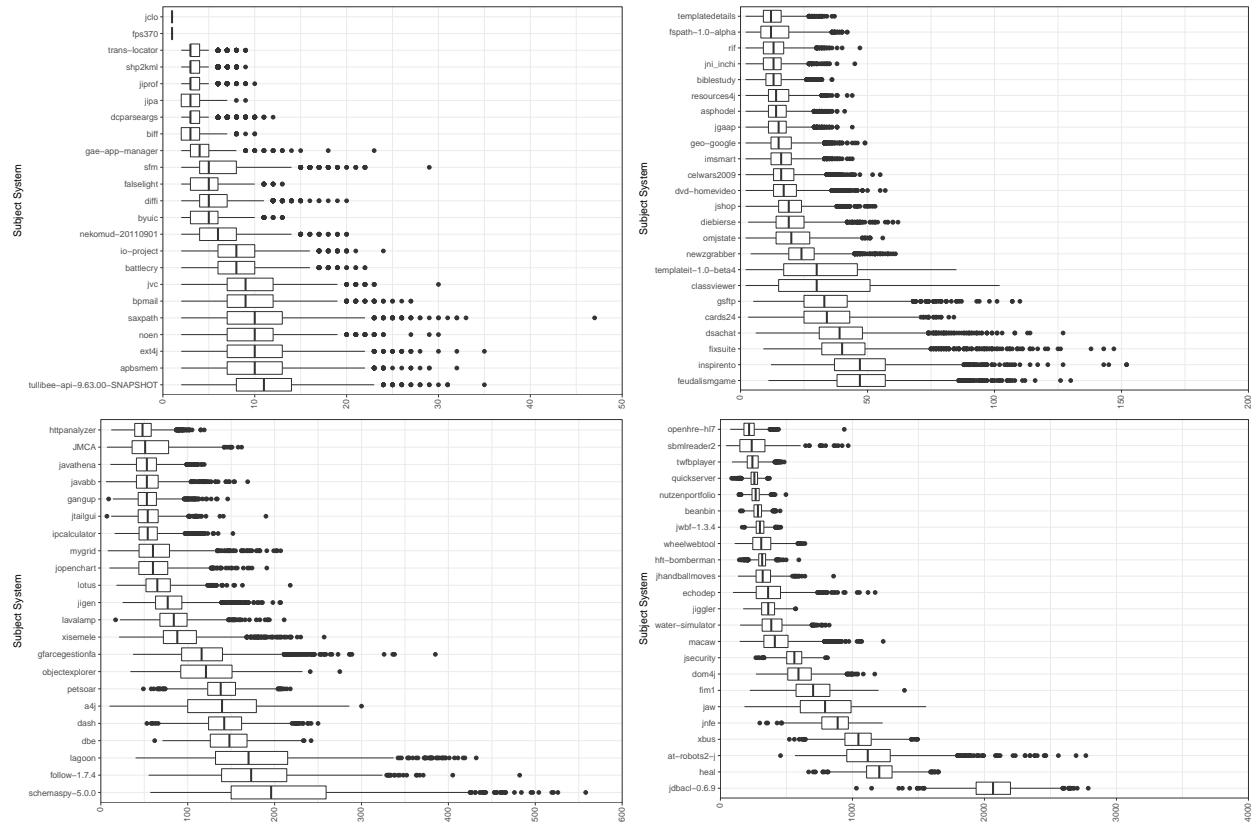
Fig. 7. Box plot of the number of SUMO iterations for convergence for each subject system for the 75% user model, showing the variance for each subject system. The breaks are to illustrate scale.
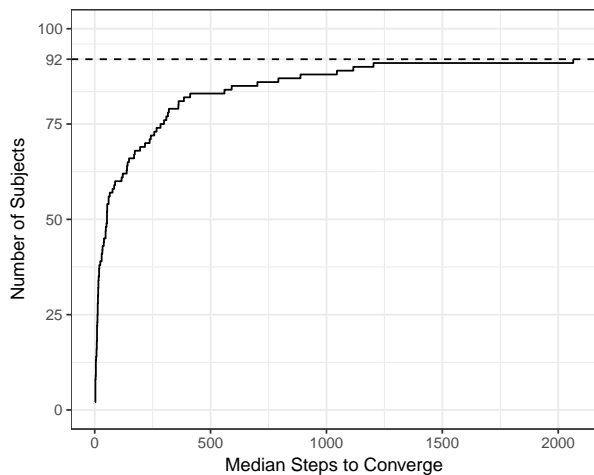

Fig. 8. Cumulative number of subject systems ordered by median steps to converge. Half of the SF100 is remodularised within 51 SUMO iterations; most are remodularised within 500. The dashed line indicates the total number of subject systems from the SF100 in the study (92)
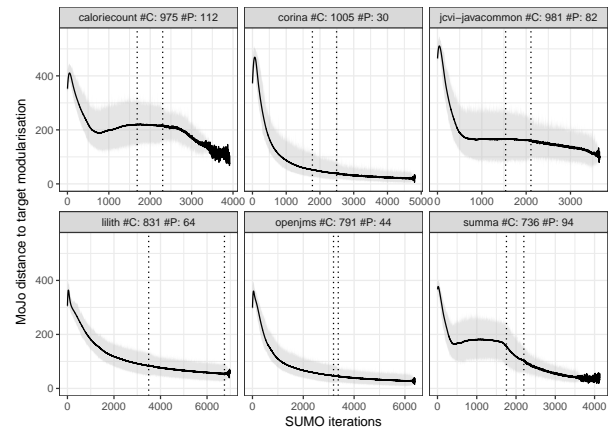

Fig. 9. The MoJo history for the 6 systems that did not converge during the time of the experiment, averaged over each recorded sample $n \geq 2,703$. The shaded background describes the range (maximum to minimum) of the recorded values at each time step. The two dashed lines indicate the maximum step number where at least 90% and 10% of the repetitions were recording data respectively. The numbers in the facet titles correspond to the size of the system in classes and packages.

The role of modules is illustrated by re-examining the results in Figure 7. For example, two of the largest systems jdbacl (261 classes) and at-robots2-j (296 classes) are of a similar size in terms of their number of classes. However, jdbacl required an average of 2067 iterations in SUMO whereas at-robots2-j required 1139 — almost half that of jdbacl. The key difference between these two

systems is the number of modules; jdbacl has 25 packages, whereas at-robots2-j has only 12.

Effect of positive/negative ratio of user-supplied relations. Figure 11 plots the median number of steps required against the number of classes in the system. The different lines represent different proportions of positive relations used to parameterise the user model. The solid line (uppermost)
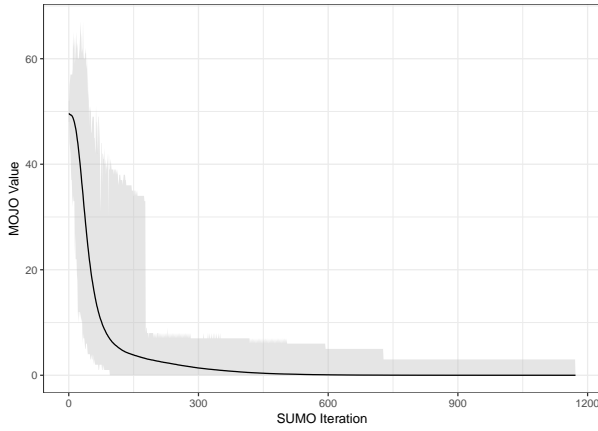
Fig. 10. Improvement (in terms of MoJo) of the solution is most rapid at the start of the refinement. The shaded region represents the minimum and maximum MoJo values across 3000 solutions at each time step. The line shows the mean at each time step.

Fig. 11. The effect of positive relations on median number of steps required by SUMO. The lines represent different ratios between positive and negative relations provided by the user model. Each point is the median calculated from 3000 samples. The plot is divided into two to preserve the scale of the data for smaller subjects.

represents the model that will always choose negative relations if possible. The dotted line (bottom) represents the model that will always choose positive relations if possible. The others represent different weightings in between.

The results show that there is a clear advantage to emphasising the selection of positive relations where possible. This advantage becomes more pronounced, the bigger the system is. Looking at the performances for systems consisting of between 100 and 150 classes, choosing to always emphasise positive edges can reduce the number of steps required by 500 to 1000. For the larger systems, the difference increases further.
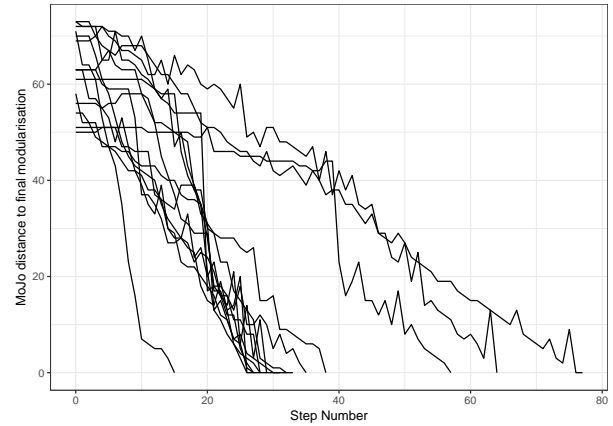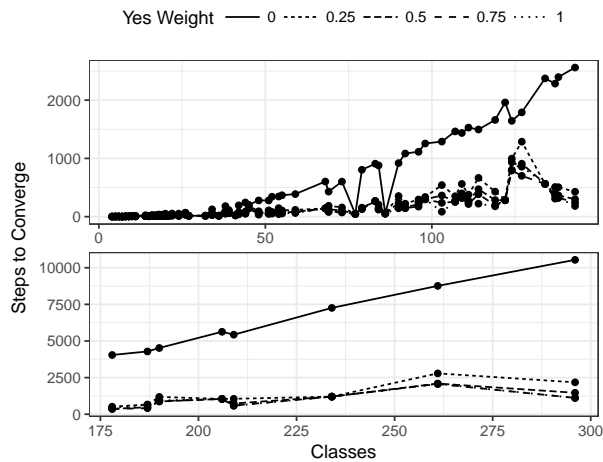

Fig. 12. Selection of traces (for users starting off from a similar initial MoJo distance) from the user-study wrt. Google Guava described in Section 4.5.

> *RQ2: The initial modularisation rarely has a significant bearing on the performance of SUMO, and only if the system is small. The size of the system, and the (ideal) number of modules in the final modularisation, have a substantial impact on the efficiency of SUMO. Providing positive relations (stating that two modules belong together) is much more effective than providing negative relations (stating that two modules don't belong together).*

5.2.3 Contrasts with the user testing sessions

One of the threats to validity discussed in Section 5.1.4 pertains to the fact that our user-model is somewhat simplistic. It selects relations between classes in a quasi-random way. This potentially leads to a performance that under-approximates the performance of an actual developer. A real user could, for example, be expected to scrutinise the current set of modules, and to pick relations between classes that specifically contradict (and therefore correct) mistakes in the modules proposed by SUMO.

Figure 12 shows a selection of traces of MoJo distances as users interacted with SUMO to remodularise Google Guava (see Section 4.5). These tend to show a more linear decrease in the MoJo score and do not exhibit the long 'tail' of minor improvements found in traces produced by the automated user (e.g. in Figure 10). Although many of the user interactions end with a small number of minor refinements, this is not nearly as pronounced as the long tail produced by the user model. This is usually due to the fact that the user model globally selects relations that are trivial, and do not necessary contradict SUMO's current hypothesis, whereas human users tend to be more judicious in the choice of feedback, choosing to target relations that contradict SUMO's hypothesis. The tail in the user model traces corresponds to points where the majority of the remaining feedback supplied was non-corrective (c.f. the Coupon Collector problem). In contrast the mix of corrective and non-corrective information is more homogenous throughout the process.

While we observed our users providing feedback methodically, often localised to smaller pieces of the system, the magnitude of the improvement between steps for the

human users is not consistent: although the overall trend is approximately linear, some iterations result in greater MoJo changes than others. The pattern between users is also less consistent, which corresponds to the different nature in which each user chose to solve the problem. The subjectivity of remodularisation also shows in the differing MoJo scores for each user trace. In our user testing, users provided between 39 and 155 relationships during the exercise (mean: 96.1, median: 103.5) relations. Five of the 35 users opted to use the lock option (Section 4.4), which on their behalf provided between 3890 and 7381 relations. Most (4) of these users chose to lock every module, while one used the feature to freeze part of their solution. The lock functionality allows users to provide a large volume of feedback with relatively little effort.

Our empirical work indicates that SUMO can scale for large constraint systems. In the context of our experiments, the largest constraint system SUMO successfully solved contained $42,808$ relationships. This is sufficient to represent every constraint for a subject system containing 293 classes (by the worst-case formula). In practice, SUMO found the target within a small fraction of this value (median = 0.266). Users judiciously applying feedback as they did in our testing are likely to reduce this factor further, and where necessary the lock button can assist them to provide an otherwise prohibitive volume of feedback. Users are free to save any solution generated, and as shown by our empirical results, intermediate solutions produced are often net improvements (in terms of MoJo score) — even for the largest systems we studied in Figure 9.

6 RELATED WORK

As mentioned in the background section, there are several existing techniques that rely upon user feedback to produce remodularisations. In Section 6.1 we discuss these approaches and contrast them with SUMO. The process by which we obtain feedback from the user is closely related to the activity of "constraint acquisition"; this relationship is explored in Section 6.2. There are further parallels between SUMO and the constraint-based architecture recovery, which are discussed in Section 6.3. Finally, Section 6.4 discusses the related work on visualising modularisations, and relates these to the SUMO graphical user interface.

6.1 Interactive Approaches

Fully interactive approaches focus on enabling the user to provide feedback as the algorithm runs. Arch [13], for example, allows the user to confirm or reject merges suggested by a hierarchical clustering algorithm. A similar approach has been used to incorporate feedback into a genetic algorithm [16], similar to Bunch [11]. In this Interactive Genetic Algorithm (IGA) process, the user is periodically asked for feedback on parts of the solution. The IGA uses the same type of constraint as SUMO; pair-wise positive or negative relations. However, the user provides feedback only where the algorithm requests it, subject to random selection, or prioritised selection based on the size of clusters, although Bavota et al. remark that their approach could be adapted

to allow the developer to choose where to provide feedback [16]. In contrast, SUMO allows the user to apply as much knowledge wherever they choose.

The IGA uses constraints to alter the fitness values of individuals used in the search process. Solutions that violate constraints are penalised according to a parameter of the fitness function. This contrasts to SUMO, which does not permit any constraints to be violated. Unlike the IGA, SUMO does not rely on any other optimisation methods in parallel with the interactive refinement. Rather, SUMO incorporates similar heuristic information only through the initial solution which is then refined subject to user feedback.

The solutions returned by SUMO are those first suggested by the constraint solver (after initialising it with a previous solution as described in Section 4.2) rather than the best according to a metric. The difference in the two approaches is that with the IGA the user sees partially optimised architectures and guides the algorithm, whereas with SUMO, the user "tunes" the architecture after an algorithm has optimised it.

Abdeen et al. [36] propose a similar approach to Bavota et al.'s interactive genetic algorithm to build a sequence of refactorings. The user can specify where changes are made a-priori, using the length of the refactoring sequence to limit the allowable changes. Similarly, Bavota et al. [37] study the use of localised remodularisation to decompose poorly composed monolithic packages into smaller components. They achieve this by looking beyond the coupling inherent in the code base by using semantic measures of coupling.

Tonella explored the tradeoff [8] associated with remodularisation, finding that some refactorings can produce a better result, but require more extensive changes. Neither Abdeen et al.'s approach nor Bavota et al.'s semantic coupling methods are automatic in the same sense as ours. Instead they rely on the user targeting areas to be renovated a-priori. In contrast, SUMO allows the user to provide feedback wherever appropriate in an online process. Furthermore, SUMO can still be applied with these localised refactoring techniques as it does not prescribe one particular method to generate the initial remodularisation to be refined, and can thus complement them.

6.2 Constraint Acquisition

The SUMO process of compiling a set of constraints with user interactions has also been approached from a query-driven perspective. CONACQ [38] poses specific queries to the user, using them as an oracle to deduce the constraint network. This contrasts to SUMO in that the queries are "membership queries" (e.g., *"do x and y belong together?"*) versus the "equivalence queries" used in SUMO (e.g., *"Is this correct? If not, why?"*) [39]. The former constraints are similar to the approach taken by Bavota et al. [16] in that they specify the elements about which information is required.

Bessière et al.'s approach [38] could feasibly be incorporated into SUMO; one problem to be addressed is the provision of redundant constraints by the users, observed and modelled in the first and second stages of our evaluation

respectively. CONACQ seeks to reduce the capture of this information by ensuring the queries posed will improve the state of the model. We intend to investigate this potential solution in future work.

6.3 Constraint Satisfaction in Architecture Recovery

Remodularisation is closely related to the process of architecture recovery. Architecture recovery processes aim to extract higher-level abstractions over the source code, but are not restricted to producing modularisations. Instead, different perspectives, or *views* for different concerns are produced [40], [41]. Constraint satisfaction has also been deployed in architecture recovery, wherein patterns are matched over the code base. These patterns may be taken from a repository [42], or specified in a query language [43].

SUMO differs in the type of view required (a grouping of modules, rather than an architectural view), and the information its constraints encode. In the pattern matching approaches, constraints define abstract structures that the code must exhibit for a view to be instantiated. SUMO views the problem principally through the connectivity of the domain knowledge given and has a very weak reliance on the original system's dependency structure (this may be implicitly introduced by the choice of remodularisation algorithm, e.g. Bunch, used to generate the starting configuration). This loss of information to SUMO is mitigated by its use as a final filtering step, designed to handle scenarios where the value of the dependency graph has been extracted by the modularisation algorithm applied in the previous step.

6.4 Visualising Modularisations

Visualising the proposed solutions is challenging. SUMO necessarily visualises the relationships between entities, supplanting this information with the existing documentation for the system.

The Distribution Map [44] allows proposed changes to be visualised at the class and package level. It is a visual analogue to similarity metrics such as MoJo [45]. A Distribution Map shows new packages containing the reorganised entities, which are coloured based on their previous assignments. SUMO, in contrast, only shows the modularisation's current assignments. However, Distribution Maps could be used to assess the final result of a SUMO refinement, or in determining if applying SUMO would be necessary, and are thus complementary to our approach.

Abdeen et al. [46] describe a visualisation method for modularisations that represents the structure of each package in the system. The Package Blueprint visualises packages as a dependency matrix, showing internal and external dependencies. This visualisation allows architectural problems to be detected at a structural level by analysing the dependencies. For example, by investigating the internal dependency structure of a package in the Blueprint, low cohesion is signified by a sparse dependency matrix. This technique allows architectural problems to be observed. This approach is similar to the Package Fingerprint [47], a technique that allows maintainers to visually inspect their architectures for similar potential defects by presenting the dependencies amongst packages in the system.

Unlike Package Blueprints and Fingerprints, SUMO does not attempt to visualise the structure of the system, making it more similar to Distribution Maps in the information it displays. SUMO instead relies on other remodularisation algorithms to rectify any architectural problems prior to refinement.

The design of SUMO is sufficiently flexible that other visualisation techniques may be applied, however. The *identifyCorrections* function given in Section 4.1 manages presentation and could be readily adapted to use an alternative visualisation to the one used in the user study, described in Section 4.4.

7 Conclusions and Future Work

In this paper, we described SUMO, an algorithm designed to solve the problems faced when using automatic remodularisation algorithms. The interactive element of SUMO creates a virtuous cycle of improvement that is guaranteed to be consistent with the user's domain knowledge. Transitivity of positive relations enables SUMO to make the most of the information provided by the user, which we demonstrated in Theorem 2 as well as practically in a comprehensive systematic evaluation.

The study, built on data collected from an observational pilot study, explored the performance of SUMO for a diverse set of software systems. The results indicate that SUMO produces accurate result from an amount of user input that falls well below the theoretical worst-case. They also indicate that in order to produce an approximate result, this can be achieved with an amount of input that is far lower than the amount required to obtain an exact result.

Our ongoing and future work is focused on further improving SUMO. This involves the incorporation of automated source code refactoring tools that work alongside SUMO [48], refining the constraint satisfaction algorithm to favour better results, and improving the user-interface to enable the user to be more expressive when specifying relations between classes and modules. Alongside these improvements, we also intend to experiment with the application of SUMO in an industrial context.

The way in which we initialise the constraint solver leaves it free to select any solution, but more likely to choose similar solutions. This initialisation could be further improved by representing the similarity to the previous solution explicitly by treating the problem as a constrained minimisation problem. This would require adoption of an appropriate objective function and is a strategy to the problem similar to Bavota et al.'s interactive GA [16]. We intend to evaluate to what extent we can improve the solutions produced by *identifyCorrections* without negatively impacting the speed at which it returns solutions.

We intend to improve the SUMO interface implementing *identifyCorrections* to make the supply of relations easier for users. Currently, a "lock" button is the only short cut to providing a large number of relations. We will implement similar features to enable users to quickly combine two modules or split them by supplying the underlying relationships on the user's behalf. Furthermore, the algorithm we describe does not allow the user to revise feedback (beyond reversing the sequence of operations that applied it via an

undo button). We seek to improve the usability of the tool by permitting arbitrary relationships to be changed at any point.

The SUMO approach is advantageous as it does not rely on the information already present in the objects to be clustered. This makes SUMO suitable in scenarios beyond remodularisation. Bunch [11] was found to be similarly versatile due to its treatment of the problem as graph partitioning [49]. A significant part of our future work involves locating problems that have similar parallels with remodularisation that SUMO may be able to solve. For instance, Tonella's application of remodularisation targeted functions [8] highlights the existence of the remodularisation problem at varying levels. While SUMO only currently takes constraints from the user, automated extraction of assistive constraints, such as those used in specifying general-purpose clustering algorithms [15] could make the SUMO framework applicable to more complex refactoring operations beyond migration of entities between modules.

Beyond these changes, we see potential in unification of automated remodularisation tools with user feedback in the SUMO framework. Bavota et al. [16] have shown that using human input can be used to constrain an optimisation process. We aim to investigate how SUMO can formalise an environment in which multiple tools make suggested remodularisation or refactoring changes subject to constraints submitted by the user or by other automated tools (for instance type checkers).

REFERENCES

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[2] M. D. 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.

[3] 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, pp. 113–141, March/April 2009.

[4] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25–30 September 2005, Budapest, Hungary*. IEEE Computer Society, 2005, pp. 525–535.

[5] N. Anquetil and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 1999, pp. 235–255.

[6] ——, "Recovering software architecture from the names of source files," *Journal of Software Maintenance*, vol. 11, no. 3, pp. 201–221, 1999.

[7] A. van Deursen and T. Kuipers, "Identifying objects using cluster and concept analysis," in *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16–22, 1999*, B. W. Boehm, D. Garlan, and J. Kramer, Eds. ACM, 1999, pp. 246–255.

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

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

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

[11] B. S. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems Using the Bunch Tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[12] N. Anquetil and J. Laval, "Legacy software restructuring: Analyzing a concrete case," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1–4 March 2011, Oldenburg, Germany*, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE Computer Society, 2011, pp. 279–286.

[13] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *Proceedings of the 13th International Conference on Software Engineering (ICSE 1991), Austin, TX, USA, May 13–17, 1991*, L. Belady, D. R. Barstow, and K. Torii, Eds. IEEE Computer Society / ACM Press, 1991, pp. 83–92.

[14] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23–28, 2012*. IEEE Computer Society, 2012, pp. 472–481.

[15] K. Wagstaff, C. Cardie, S. Rogers, and S. Schrödl, "Constrained k-means clustering with background knowledge," in *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, C. E. Brodley and A. P. Danyluk, Eds. Morgan Kaufmann, 2001, pp. 577–584.

[16] G. Bavota, F. Carnevale, A. D. Lucia, M. D. Penta, and R. Oliveto, "Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization," in *Search Based Software Engineering - 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28–30, 2012. Proceedings*, ser. Lecture Notes in Computer Science, G. Fraser and J. T. de Souza, Eds., vol. 7515. Springer, 2012, pp. 75–89.

[17] L. Ponisio and O. Nierstrasz, "Using context information to re-architect a system," *Proceedings of the 3rd Software Measurement European Forum*, vol. 2006, pp. 91–103, 2006.

[18] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6–10, 2011*, P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds. IEEE, 2011, pp. 552–555.

[19] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Transactions on Software Engineering Methodology*, vol. 25, no. 3, pp. 24:1–24:28, 2016. [Online]. Available: http://doi.acm.org/10.1145/2928268

[20] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, dec 1993.

[21] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang, "Multiple layer clustering of large software systems," in *12th Working Conference on Reverse Engineering (WCRE 2005), 7–11 November 2005, Pittsburgh, PA, USA*. IEEE Computer Society, 2005, pp. 79–88.

[22] A. Marx, F. Beck, and S. Diehl, "Computer-aided extraction of software components," in *17th Working Conference on Reverse Engineering, WCRE 2010, 13–16 October 2010, Beverly, MA, USA*, G. Antoniol, M. Pinzger, and E. J. Chikofsky, Eds. IEEE Computer Society, 2010, pp. 183–192.

[23] G.-C. Rota, "The Number of Partitions of a Set," *The American Mathematical Monthly*, vol. 71, no. 5, pp. 498–504, May 1964.

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

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

[26] N. Jussien, G. Rochart, and X. Lorca, "The CHOCO constraint programming solver," in *CPAIOR'08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008.

[27] J. Heer, S. K. Card, and J. A. Landay, "Prefuse: a toolkit for interactive information visualization," in *Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2–7, 2005*, G. C. van der Veer and C. Gale, Eds. ACM, 2005, pp. 421–430.

[28] 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, 1999.

[29] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland*. IEEE, 2012, pp. 178–188.

[30] B. Kitchenham, S. L. Pfleeger, L. Pickard, P. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for

empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, 2002. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2002.1027796

[31] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org/

[32] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis," *Journal of the American Statistical association*, vol. 47, no. 260, pp. 583–621, Dec. 1952.

[33] A. Arcuri and L. C. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification Reliability*, vol. 24, no. 3, pp. 219–250, 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1486

[34] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[35] 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.

[36] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse, "Towards automatically improving package structure while respecting original design decisions," in *20th Working Conference on Reverse Engineering (WCRE 2013), 14–17 October 2013, Koblenz, Germany*. IEEE Computer Society, 2013.

[37] G. Bavota, A. Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, pp. 901–932, Oct. 2012.

[38] C. Bessière, R. Coletta, B. O'Sullivan, and M. Paulin, "Query-driven constraint acquisition," in *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6–12, 2007*, M. M. Veloso, Ed., 2007, pp. 50–55.

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

[40] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva, "Symphony: View-driven software architecture reconstruction," in *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12–15 June 2004, Oslo, Norway*. IEEE Computer Society, 2004, pp. 122–134.

[41] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[42] S. G. Woods and Q. Yang, "Program understanding as constraint satisfaction: Representation and reasoning techniques," *Automated Software Engineering*, vol. 5, no. 2, pp. 147–181, 1998.

[43] K. Sartipi, K. Kontogiannis, and F. Mavaddat, "A pattern matching framework for software architecture recovery and restructuring," in *8th International Workshop on Program Comprehension (IWPC 2000), 10–11 June 2000, Limerick, Ireland*. IEEE Computer Society, 2000, pp. 37–47.

[44] S. Ducasse, T. Gîrba, and A. Kuhn, "Distribution map," in *ICSM*. IEEE Computer Society, 2006, pp. 203–212.

[45] Z. Wen and V. Tzerpos, "An optimal algorithm for mojo distance," in *11th International Workshop on Program Comprehension (IWPC 2003), May 10–11, 2003, Portland, Oregon, USA*. IEEE Computer Society, 2003, pp. 227–235.

[46] H. Abdeen, S. Ducasse, D. Pollet, I. Alloui, and J.-R. Falleri, "The package blueprint: Visually analyzing and quantifying packages dependencies," *Science of Computer Programming*, vol. 89, pp. 298–319, Sep. 2014.

[47] H. Abdeen, S. Ducasse, D. Pollet, and I. Alloui, "Package fingerprints: A visual summary of package interface usage," *Information & Software Technology*, vol. 52, no. 12, pp. 1312–1330, 2010.

[48] M. Hall, M. A. Khojaye, N. Walkinshaw, and P. McMinn, "Establishing the source code disruption caused by automated remodularisation tools," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 466–470.

[49] M. B. Cohen, S. B. Kooi, and W. Srisa-an, "Clustering the heap in multi-threaded applications for improved garbage collection," in *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8–12, 2006*, M. Cattolico, Ed. ACM, 2006, pp. 1901–1908.

Mathew Hall Mathew Hall is a Research Associate in the Department of Computer Science at the University of Sheffield. His research interests focus on applications of machine learning and meta-heuristics to software maintenance problems, including testing and reverse engineering.

Neil Walkinshaw Neil Walkinshaw is an Associate Professor in the Department of Informatics at the University of Leicester. His research is mainly concerned with the application of Automated Reasoning and Machine Learning techniques to Software Engineering tasks such as testing and reengineering.

Phil McMinn Phil McMinn is a Reader (Associate Professor) in the Department of Computer Science at the University of Sheffield, where he has been researching and teaching software engineering since 2006. His main research interests include search-based software engineering, software testing, program transformation, and reverse engineering.