Improving Software Remodularisation

Mathew James Hall

Submitted for the degree of Doctor of Philosophy

Department of Computer Science

March 2013

Supervisor: Dr Phil McMinn

University of Sheffield

Abstract

Maintenance is estimated to be the most expensive stage of the software development lifecycle. While documentation is widely considered essential to reduce the cost of maintaining software, it is commonly neglected. Automated reverse engineering tools present a potential solution to this problem by allowing documentation, in the form of models, to be produced cheaply.

State machines, module dependency graphs (MDGs), and other software models may be extracted automatically from software using reverse engineering tools. However the models are typically large and complex due to a lack of abstraction. Solutions to this problem use transformations (state machines) or "remodularisation" (MDGs) to enrich the diagram with a hierarchy to uncover the system's structure.

This task is complicated by the subjectivity of the problem. Automated techniques aim to optimise the structure, either through design quality metrics or by grouping elements by the limited number of available features. Both of these approaches can lead to a mismatch between the algorithm's output and the developer's intentions. This thesis addresses the problem from two perspectives: firstly, the improvement of automated hierarchy generation to the extent possible, and then augmentation using additional expert knowledge in a refinement process.

Investigation begins on the application of remodularisation to the state machine hierarchy generation problem, which is shown to be feasible, due to the common underlying graph structure present in both MDGs and state machines. Following this success, genetic programming is investigated as a means to improve upon this result, which is found to produce hierarchies that better optimise a quality metric at higher levels.

The disparity between metric-maximising performance and human-acceptable performance is then examined, resulting in the SUMO algorithm, which incorporates domain knowledge to interactively refine a modularisation. The thesis concludes with an empirical user study conducted with 35 participants, showing, while its performance is highly dependent on the individual user, SUMO allows a modularisation of a 122 file component to be refined in a short period of time (within an hour for most participants).

Acknowledgements

Gratitude is firstly owed to my supervisor, Dr Phil McMinn for his tireless and continued support, without which I would not have been able to make the achievements detailed in this thesis. I would also like to thank Dr Neil Walkinshaw for his contributions to the joint work undertaken throughout my PhD and for the invaluable advice he has given me throughout my project. Thanks also go to Dr Graham Birtwistle for his advice and support during both my time in Sheffield as both an undergraduate and postgraduate. Gratitude is also owed to Dr Kirill Bogdanov for his work that led to the start of this PhD and his input to my research at the early stages. Additionally, I would like to thank Dr Steve Swift for his willingness to share some of the case studies used in this work. I would also like to thank the other members of the Verification and Testing research group for creating such an interesting, inspiring, and supportive environment in which to conduct research. I would also like to thank Dr Henry Addico, Ciprian Dragomir and Dr Ramsay Taylor for their support and their encouragement to take a break every now and then.

Finally, my partner Kat, and my family are deserving of a million thanks for their amazing support and tolerance throughout my postgraduate degree at Sheffield.

My PhD was supported by the Engineering and Physical Sciences Research Council under the REGI grant, number EP/F065825/1. I am eternally grateful for this, and the otherwise inaccessible opportunities and avenues it has enabled me to pursue.

Declaration

The work presented in this thesis is original work undertaken between October 2009 and December 2012 at the University of Sheffield. Pieces of this work have been published elsewhere:

- M. Hall, P. McMinn, and N. Walkinshaw. 2010. **Superstate identification for state machines using search-based clustering.** In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2010, Portland, Oregon, USA, July 7-11, 2010). ACM, 1381–1388.
- M. Hall. 2011. **Search based hierarchy generation for reverse engineered state machines.** In Proceedings of SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011. ACM, 392–395.
- M. Hall, N. Walkinshaw, and P. McMinn. 2012. **Supervised Software Remodularisation.** In Proceedings of the 28th International Conference on Software Maintenance (ICSM 2012), Riva del Garda, Trento, Italy, 472–481.
- M. Hall, and P. McMinn. 2012. **An Analysis of the Performance of the Bunch Modularisation Algorithm's Hierarchy Generation Approach.** In Fast Abstracts of the 4th Symposium on Search Based Software Engineering (SSBSE 2012), Riva del Garda, Trento, Italy, 19–24.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software maintenance benefits from the presence of high quality up-to-date documentation. In a survey of software engineers, Lethbridge et al. [64] found that while documentation was considered useful, it was also likely to be neglected and in some cases untrustworthy. In "Software Aging", Parnas [89] warns that documentation is mandatory to facilitate maintenance, attributing the all-too-common neglect of documentation to the focus on short term goals. Documentation serves as supplementary information that provides abstraction, a fundamental tool in software engineering for the management of complexity.

§ 1.1 Reverse Engineering

In cases where software documentation is lacking or requires maintenance, it must be "reverse engineered" from the original code (and documentation) of the system. Rather than adding to the code and behaviour of the system, the behaviour is distilled from the code to gain additional knowledge about the system. Chikofsky and Cross outlined the possible desired outcomes of a reverse engineering exercise, which include management of complexity, construction of alternate views and abstraction, as well as facilitation of reuse, recovery of lost information and detection of side effects [20].

For large systems, the process can be costly as it requires the reverse engineer to acquire an understanding of the code-base before documentation can be produced. This difficulty has given rise to automated and semi-automated techniques designed to make the reverse engineering process easier, and therefore cheaper.

Reverse engineering tools allow code and data to be analysed to gain insights into the software. One of Chikofsky and Cross's goals, construction of alternate views, can be achieved by creating models that represent aspects of the software. Behaviour inference, for example, can be used to automatically construct a behaviour model for software from data gathered using dynamic analysis [123]. Various forms of models can be extracted from the codebase, although this thesis applies only to two types of model that represent the software as a graph: state machines and module dependency graphs.

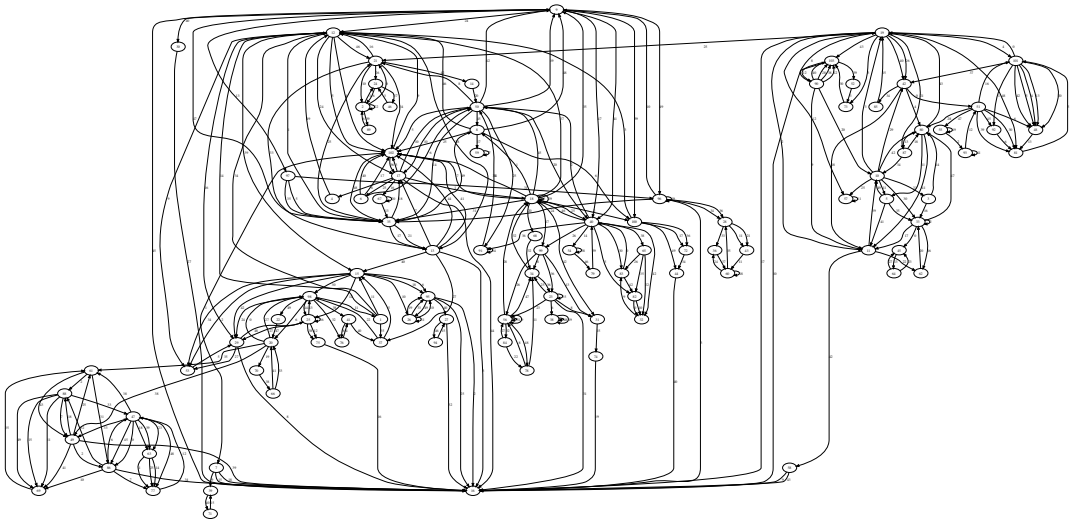


Figure 1.1: An example of a large complex finite state machine

1.1.1 Representing Software as a Graph

Graphs provide a versatile structure which can be used to represent various aspects of a system. In both of the model types studied in this thesis, the nodes in each graph represent entities, and edges between them encode some form of relation. The semantics of the relation, and the data represented by nodes varies between the two models. In addition to both being graphs, both of these model types can be extracted automatically through program analysis. The following subsections introduce state machines and dependency graphs.

State Machines

State machines represent the behaviour of a system, at a level above the implementation details and are instances of labelled transition systems (LTS). LTSs are general encodings of behaviour in the form of transitions between states [57]. State machines have been used in various forms to encode specifications [44, 50] as well as in model checking. State-based models are advantageous for program comprehension due to the abstraction provided. Additionally, they can be derived from software automatically (and therefore cheaply) [5, 25, 122, 123].

Figure 1.1 depicts a large state machine. Although this example may represent the behaviour at a level above the source code, it is still large. This issue of size and lack of abstraction forms the basis of the problem that this thesis investigates.

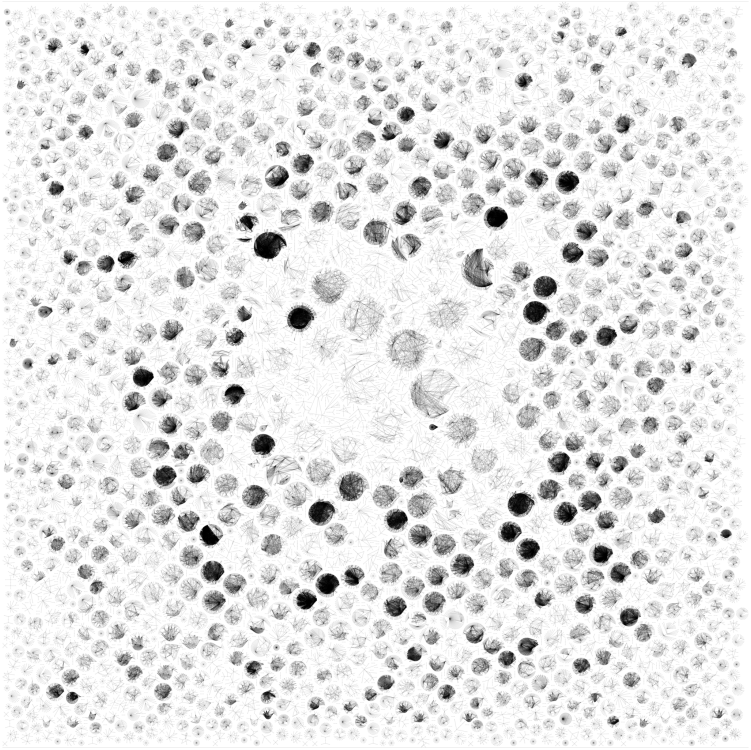


Figure 1.2: The MDG for the Linux kernel 3.2

Module Dependency Graphs

Dependency graphs of software systems model the interaction of their components (at a broader level of detail than behavioural models). They represent the overall organisation of entities in the system. A system may have multiple dependency graphs, constructed for various types of entities. For example files have dependencies [9], as do variables [111], compound graphs that include various types of entities in one have also been studied [83], although the focus of this thesis is on module dependency graphs, constructed at the source file level.

Module dependency graphs (MDGs) are directed or undirected graphs, files in the system are vertices in the graph. Edges are constructed for each dependency one file has on another. Figure 1.2 shows the MDG of all C source files in version 3.2 of the Linux kernel, constructed from the include directives contained within each file. The complexity of such a large MDG makes it impossible to manage without additional organisation; the example comprises $30,443$ C source and header files.

1.1.2 Complexity of Flat Graph-based Representations

Although these models provide a view of the system which abstracts away a large amount of the implementation, the models are often large. With state machines reverse engineered from software, the size is determined by the complexity of the software and

the granularity at which it is instrumented. The size of module dependency graphs are determined by the size of the software project itself.

The low management of complexity of these representations becomes more problematic as the system size increases. Similarly, the requirement for comprehension aids may also increase with system size, leading to a point where even the abstraction offered by these models is insufficient to achieve Chikofsky and Cross [20]'s goals of providing alternate views, managing complexity or abstraction.

Moreover, overly complicated documentation which is hard to consult for information is at risk of being ignored by developers. Lethbridge et al. [64] observed that developers felt "finding useful content in documentation can be so challenging that people might not try to do so", which emphasises the importance of clear documentation.

This size problem has resulted in the use of extensions to the models to manage their complexity. The following subsection focuses on hierarchical abstractions that have been used for this purpose.

1.1.3 Managing Complexity with Hierarchies

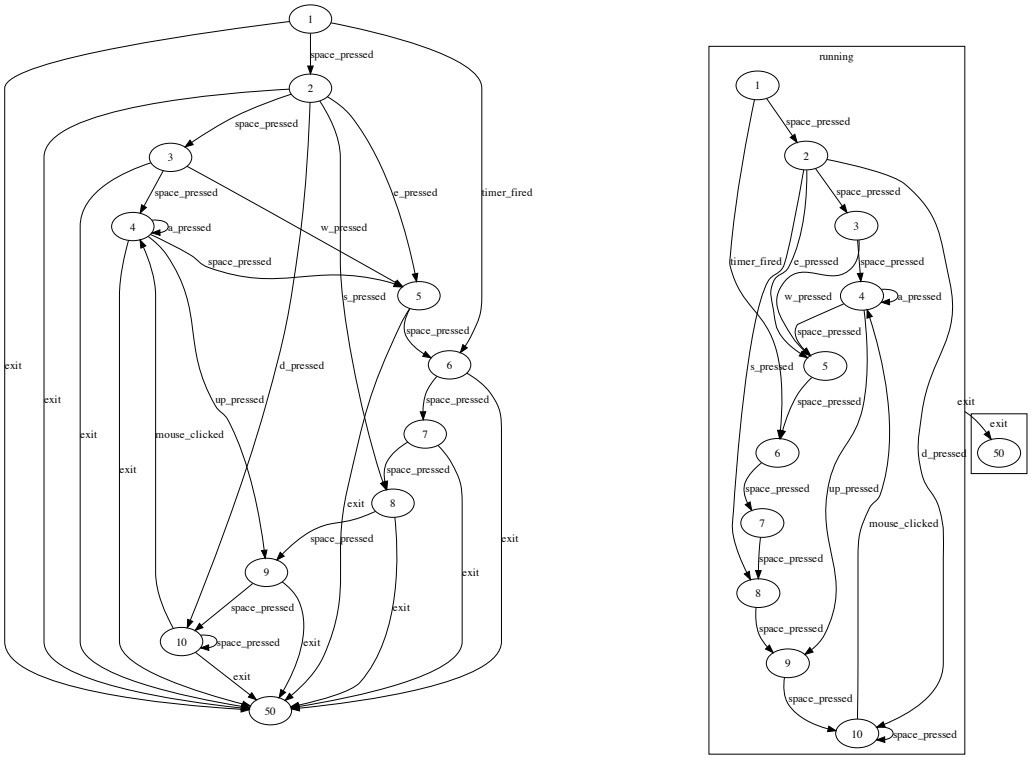
One method used to manage complexity in large models is the addition of grouping or hierarchies of entities. Hierarchies provide additional organisation and allow higher levels of abstraction and can be applied to both state machines and dependency graphs. Statecharts, for example define encapsulating "composite" states for state machines [44] and the filesystem, or packages, may be used to group files in a dependency graph. The following subsections give examples of these additions, and a more detailed assessment of these model variants and techniques for producing them appears in Chapter 2.

Statecharts

Harel's Statechart formalism [44], a prevalent type of state-based model, expresses hierarchies of behaviour for state machines. Statecharts extend state machines with a new type of state, a "superstate", which is comprised of one or more states. Superstates may themselves encapsulate other superstates, allowing arbitrary depths of nesting of behaviour. Abstraction is provided by a superstate, which implements a general behaviour, the specifics of which are encoded in the states it encapsulates.

In addition to a hierarchy of states, statecharts allow transitions from and to superstates. These represent common transitions to or from all the states within a superstate. Subsumed transitions allow the overall edge count to be reduced, which reduces the visual complexity of the diagram. This is visible in the example in Figure 1.3. A large number of the edges in part **(a)** of the diagram are duplicates of the "exit" transition. Part **(b)** shows the same diagram with the addition of two superstates and the subsumption of the common transitions.

The example in Figure 1.3 depicts only one of many possible groupings of states into two superstates; there is not one canonical statechart for a given state machine. The grouping is not necessarily "mechanical", and may include subjective judgements. Work



(a) An example state machine with many redundant edges.

(b) A statechart of example **(a)**.

Figure 1.3: A state machine **(a)** and its statechart equivalent **(b)**

exists which aims to automate this process to an extent using rule-based transformation operations; these approaches are discussed further in in the literature review in Section 2.4.

Module Dependency Graphs

In MDGs, a hierarchy (or "modularisation") can be introduced for the benefit of grouping common functionality into components. This partitioning is normally performed as an organisational step by developers; some languages provide a means to expose a hierarchy of groups in the form of nesting packages (Java [41, p. 163]) or namespaces (C++ [110, p. 152]).

Using the same Linux kernel example from Section 1.1.1, the $30,443$ source files are organised into $2,171$ directories and subdirectories. These group components into larger and larger groups. For example, the top level consists of directories for architecture-specific code, which is divided into directories for each supported architecture.

As with statecharts, the package structure for a given MDG is a product of the maintainer's subjective judgements, potentially obfuscated by neglect. This makes an au-

tomatic "remodularisation" process difficult, as one canonical hierarchy may not exist for a dependency graph. This difficulty is illustrated in Wu et al.'s [126] discovery of poor recovery of original structures obtained by several remodularisation algorithms.

Approaches to automating the remodularisation process have primarily sought to apply machine learning techniques, particularly clustering [6, 9, 100, 111, 118, 125]. These approaches, discussed in depth in Section 2.5, aim to use information extracted from the software as "features" which are used to estimate how similar the elements in the dependency graph are in order to produce a grouping.

The features are used to approximate human knowledge with the intention of producing a clustering which matches, or is highly similar to that produced by an expert. Other approaches have used structural indicators based on software metrics, such as coupling [127], to estimate the "design quality" of the result [81], based on the hypothesis that a hierarchy maximising such a metric will be an acceptable solution to the problem, even if it does not match the solution produced by an expert developer.

This section has outlined the various approaches to management of complexity for the type graph-based software models discussed in the thesis. The following section further defines the problem addressed in this work, drawing on the similarity of the representations and the identical desired results of approaches to the problem.

§ 1.2 The Problem of this Thesis: Hierarchy Generation for Graph-based Software Models

This thesis investigates the similar problems of state machine hierarchy generation and software remodularisation. Despite the similarity and the equality of the underlying graph representation of both state machines and MDGs, there is a stark contrast between approaches to the two problems. In both cases, a major component is the difficulty in the determining quality of a result, both from an evaluation perspective as well as from a clustering perspective. The large search space of all possible hierarchies for large graphs also requires additional consideration; there are in excess of one billion possible partitions for a 15 node MDG [71].

The hierarchy generation problem presents two challenges. The first difficulty is in producing an algorithm or technique that is able to produce hierarchies which are as close as possible to those produced by developers. This relies on the selection of the correct set of features, algorithm and assessment method for state machines. The Bunch graph-based clustering algorithm [81] is investigated for this purpose, due to its versatility. A detailed justification for this choice is given after the literature survey, in Section 2.6.

The second challenge is the issue already faced by existing remodularisation and hierarchy generation approaches. Selection of the best set of features and best clustering algorithm is still unlikely to reliably reproduce the decomposition produced by an expert, due to the subjective nature of the task. The lack of available features beyond the structure of the graph (typical for reverse engineered state machines) further compli-

cates the matter. Methods that can enable this subjective information to be captured, while allowing the non-subjective elements of the remodularisation task to be left to an automated approach are also investigated in this thesis.

§ 1.3 Aims and Objectives

This thesis is concerned with automatic or semi-automatic remodularisation of graph-based software models, with a focus on state machines and MDGs. The goal of this process is to produce meaningful abstractions that manage the complexity of the model. This goal is comprised of two aims:

1 To apply remodularisation to the hierarchy generation problem for state-based software models; and

2 To analyse and address qualitative issues with the output of existing automated remodularisation algorithms.

The aims of the thesis are achieved through the following objectives, which form the structure on which the contributions made by the thesis are based:

1 Measure the application and analysis of an existing remodularisation technique to state-based software models

2 Produce enhancements to the state of the art of remodularisation by extending it to hierarchies

3 Develop and empirically evaluate an improved remodularisation approach through a realistic experiment with human participants

§ 1.4 Organisation and Contributions of the Thesis

Figure 1.4 maps out the structure of the investigation portrayed in this work, starting from the exploration of existing solutions to the problem, including demonstration of the similarity of the individual problems of state machine hierarchy generation and remodularisation in the literature review in Chapter 2. The first experiments in Chapter 3 explore the application of the Bunch [81] algorithm to state machines, identifying three problems which are then addressed in the subsequent research chapters. Issues with the search approach taken in Bunch are addressed in Chapter 4, while Chapters 5 and 6 address the accuracy problem, using domain knowledge.

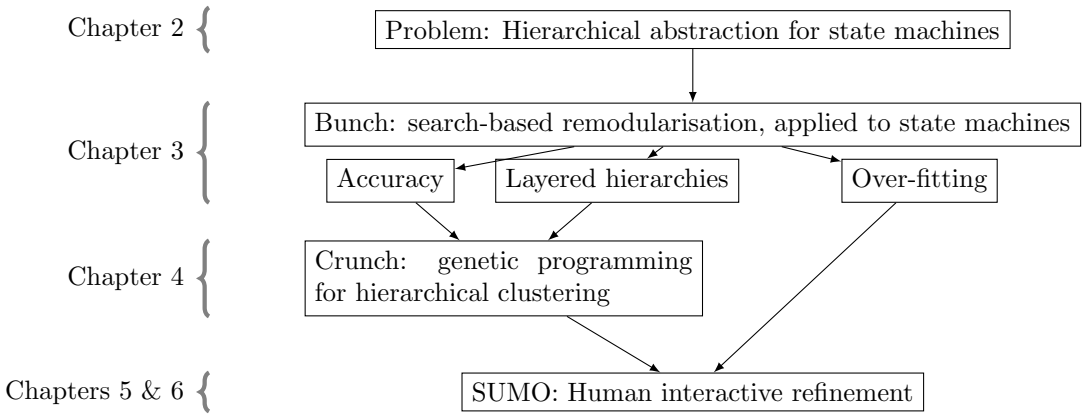


Figure 1.4: The path of investigation taken in the thesis

Chapter 2: Literature Review This chapter reviews the literature relevant to the problem. It begins by expanding upon the models used and the various extensions which introduce hierarchies to them. The similarities of hierarchy generation for both state machines and dependency graphs is demonstrated and the problem domain of the thesis of clustering graphs is defined. Hierarchy generation techniques for both state machines and MDGs are summarised, including the means used to assess them.

Chapter 3: Remodularisation of State Machines In this chapter, a proof-of-concept is given for the remodularisation of state machines with the Bunch [78] remodularisation tool and an evaluation is used to demonstrate that Bunch is capable of reproducing parts the developer-designed hierarchy for a state machine, although it does not do so perfectly.

The performance of Bunch is then further explored from the hierarchy generation perspective. A survey of several software systems demonstrates that hierarchies are rarely layered. This, combined with evidence found for over-fitting in Bunch leads to an open problem: full, variable-depth, hierarchy generation for state machines using clustering.

One approach of rectifying the problem using random mutations of Bunch hierarchies is also evaluated as a means to transform layered hierarchies into unlayered versions to improve an objective metric.

The *contributions* made by the chapter are as follows:

1 An empirical study into the feasibility of applying the Bunch search-based remodularisation tool to state machines, showing that Bunch is able to recover part of the structure of a flattened statechart.

2 Analysis of the Bunch clustering tool's layered approach to hierarchy generation, demonstrating that it causes over-fitting.

3 Investigation of the use of random mutation to produce non-layered hierarchies from layered solutions produced by Bunch to improve their quality, assessed by the mean of a quality metric.

Chapter 4: Search-based Hierarchical Clustering This chapter explores the hierarchy generation problem for search-based remodularisation. It applies several representations that enable global searching of all *hierarchies* to avoid the over-fitting issue found in Bunch. These representations are based on a transformation encoding as applied to search-based refactoring, stack-based genetic programming and a label assignment approach similar to that used in the Bunch genetic algorithm.

Evaluation of these approaches show that they can produce partitionings, although fitness values located are lower than those produced by Bunch. A further hierarchical study with the transformation encoding shows that the global search can produce hierarchies with a greater fitness at higher levels in the hierarchy than Bunch, however Bunch better optimises fitness values at the lower levels.

Objective functions are then compared using this approach, finding that two remodularisation fitness functions do not identify superstates that enable superstate transitions to be used, and that an information-theoretic fitness function [69] produces the fewest number of clusters and edges between clusters.

The following *contributions* are made:

1 An application of a genetic programming approach based on transformations that is able to represent non-layered hierarchies and explore the whole space of hierarchies in one search, unlike Bunch.

2 Evaluation of this approach for clustering MDGs and state machines, determining that it produces higher fitness values at higher levels in the hierarchy than Bunch, at a cost of overall fitness.

3 A study of the use of several metrics as fitness functions for hierarchical clustering, determining that remodularisation fitness functions reduce the number of edges leaving or entering superstates, but do not identify solutions that reduce the edge count through superstate transitions.

Chapter 5: SUMO: Supervised Software Remodularisation This chapter describes the SUMO refinement algorithm, which improves on an existing modularisation using user-supplied corrections. Equations for the amount of required domain knowledge are derived, then simulated authoritative decompositions are used to evaluate the performance of the SUMO algorithm, finding that improvement can be obtained in a short number of iterations, before convergence occurs. The chapter makes the following *contributions*:

1 A novel application of constraint-based interactive refinement to the remodularisation problem, with equations derived for the upper bound of input required to completely refine a modularisation.

2 An empirical analysis of the performance of this clustering technique, showing how improvement is observed at almost each interactive iteration of the algorithm.

Chapter 6: Human Factors Affecting SUMO This chapter presents an interactive GUI implementation of the SUMO algorithm, including enhancements made as a result of a pilot study. The performance of this modified algorithm and the tool are then evaluated in a user study, finding that all 35 of the participants were able to use the SUMO tool in a practical scenario to complete the remodularisation of a 122 file software component. The chapter makes the following *contributions*:

1 An implementation of this algorithm in an interactive tool, incorporating improvements to the SUMO to address issues identified by a pilot study.

2 Evaluation of the constraint-based remodularisation approach conducted with 35 participants, finding that the majority (23) managed to use the tool to refine a modularisation of 122 files in under an hour.

Chapter 7: Conclusions & Future Work This chapter summarises the work in the thesis and outlines potential avenues for further inquiry, including approaches to further develop the Crunch and SUMO tools presented in this thesis.

The next chapter starts by discussing the problem in greater detail, including the specifics of the models examined in this thesis and the difficulties faced by automated hierarchy generation techniques. It then shows how similarities the MDG remodularisation problem and the state machine hierarchy generation problem may enable modularisation techniques to be applied to state machines. Metaheuristic algorithms and genetic programming are also introduced, which are utilised in the Crunch tool described and evaluated in Chapter 4.

Chapter 2

Literature Review

This chapter discusses the state machine hierarchy generation problem in greater detail and summarises approaches representing the state of the art. It also introduces module dependency graphs and the remodularisation problem and demonstrates how techniques for remodularisation are applicable to state machines, united by the common graph representation used.

The discussion hierarchy generation approaches used for both problems begins with an overview of the various aspects of performance of these approaches that have been evaluated in the literature. The remainder of the chapter focuses firstly on solutions to the state machine hierarchy generation problem and then the remodularisation problem, including an introduction to clustering and metaheuristic algorithms.

Finally, the chapter concludes by summarising the techniques used and details the open research problems, before defining the elements to be investigated, starting from the application of the Bunch [81] software remodularisation tool to the state machine hierarchy generation problem.

§ 2.1 Graph Representations of Software

Software can be modelled using a variety of representations. Many graphical models use a notion of connectivity between entities to represent components of the system. For example, UML [85] defines activity diagrams which model the sequence of interactions between components over time, state diagrams which represent behaviour, data flow diagrams describe how data moves through a system and inheritance diagrams summarise the class hierarchy of the software. All of these diagrams are directed graphs.

This thesis focuses on two of these graph-based representations in particular: state-based models and module dependency graphs. This section introduces these models, and further defines the complexity problem that applies to them as their sizes increase. It concludes with a summary of the similarities of the representations, and shows how the problems of generating abstractions for each of these models are highly similar.

2.1.1 State Machines

State machines represent software as a set of states, each of which has behaviour associated with it. The system moves from its current state to another if there is a defined "transition" between them, and an appropriate event occurs to exercise it. For example, inserting a coin into a vending machine may take it from the "insert coin" state to the "make selection" state.

The most basic form of state machine is encoded in a labelled transition system (LTS). An LTS [57], is a tuple (Q, L, T) where Q is the (possibly infinite) set of states and L is the finite alphabet of labels for the transitions. The set T encodes the transitions, such that $T : Q \times L \times Q$ — a transition $t \in T$ $(a, exit, b)$ corresponds to a transition between state a and b with the label "exit". An initial state may be specified, $q_0 \in Q$, making the LTS a "rooted" transition system. LTSs may also encode non-determinism, where multiple transitions exist from the same state with the same label.

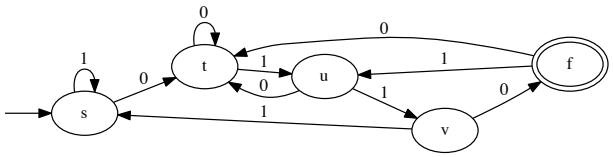


Figure 2.1: A state machine that accepts strings ending in 0110

Finite state machines (FSMs) have a finite number of states and, like LTSs, may be deterministic or non-deterministic. It is possible to transform an FSM into an LTS [92]. FSMs may be further extended to include an additional set of "final" (or "accept") states and an initial state.

Figure 2.1 shows an example state machine. In this example: $Q = \{s, t, u, v, f\}$, $\Sigma = \{0, 1\}$, $q_0 = s$, $F = \{f\}$. Only inputs ending with 0110 result in the machine remaining in the f state, the only state in the set of accept states F, thus the machine only accepts strings ending with that suffix.

It is possible to generate a state machine through dynamic analysis of a target software system [121]. A model learning algorithm can be applied to a set of traces generated by instrumentation and running the target software. One approach to the problem is state merging [33, 16, 87, 63], where a state machine containing all traces (known as a prefix tree acceptor) is reduced by iteratively merging states which are determined to be equal. This equality test relies on a source of information, which can include a heuristic measure of equality based on common tails, or use a human oracle, for example. A comparative study that further describes model learning algorithms was conducted by Ali et al. [3].

The use of dynamic analysis means the initial source of information for the machines is not abstracted at all. By tracing at the method call level, for example using AspectJ [59] or Pin [67], the resultant state machine will only model the sequence of

method calls of the system. This low level of abstraction is a problem where the machines are used for program comprehension [122]. One suggested solution to this problem is the use of manually generated "trace abstractions" which map a method call sequence to a higher level action. For example, in a drawing application multiple calls to `move_cursor, draw_point` can be abstracted to the more general `draw_line` event. These abstractions rely on the presence of an oracle, however. Extended finite state machines (EFSMs) [18], which incorporate guards on transitions have also been produced using grammar inference [66] in order to capture information not present in FSM models.

2.1.2 Hierarchical Extensions to State Machines

The limited notation of state machines has been extended in various ways to provide more expressive power for state-based models. One common extension is the addition of a hierarchy to the model [4, 44]. This allows the lower-level behaviour to be grouped together and provides abstraction through encapsulation and information hiding. The most prevalent of these state-based models is Harel's Statecharts [44], a variant of which forms part of the UML standard [85, p525]. Statecharts also extend state-based models with events on transitions, guard conditions and parallel processes, although these additions are beyond the scope of the thesis and are not discussed.

The use of encapsulating (or "composite") states also allows transitions to be rewritten where each state within the composite state has a transition with the same target and label. In doing so, the visual complexity of the diagram can be reduced.

2.1.3 Module Dependency Graphs

Module dependency graphs model the network of dependencies between files or classes in a software system. When viewed as a graph, the set of vertices V is the set of all modules, and each dependency between two files is an edge in the graph. Dependency graphs can be extracted from software using static analysis tools such as Dependency Finder [109] or Source Navigator [106]. Figure 2.2 shows an example MDG for the Linux kernel, constructed by parsing the include statements present in each source file.

The example demonstrates the necessity for organisation of MDGs. Although the elements in the graph are grouped together into "communities", there is little abstraction provided in this graph; it contains a total of $30,443$ nodes. It is, however, common for files in software projects to be partitioned in a tree structure. Java and C++ both provide functionality to group classes together in packages and namespaces respectively, and C allows source files to be placed in subdirectories in the filesystem.

The partitioning of the source files is performed by the system's developers and allows the files to be organised to facilitate navigation of the project. To illustrate this, Figure 2.3 shows the arrangement of all directories in the Linux kernel's source tree. This tree of $2,171$ directories groups all the source files in the previous MDG by related concepts. For example, there is one high-level directory which contains drivers, and another for processor-specific files. The partitioning of the source files can be applied to a module dependency graph to provide hierarchical structure.

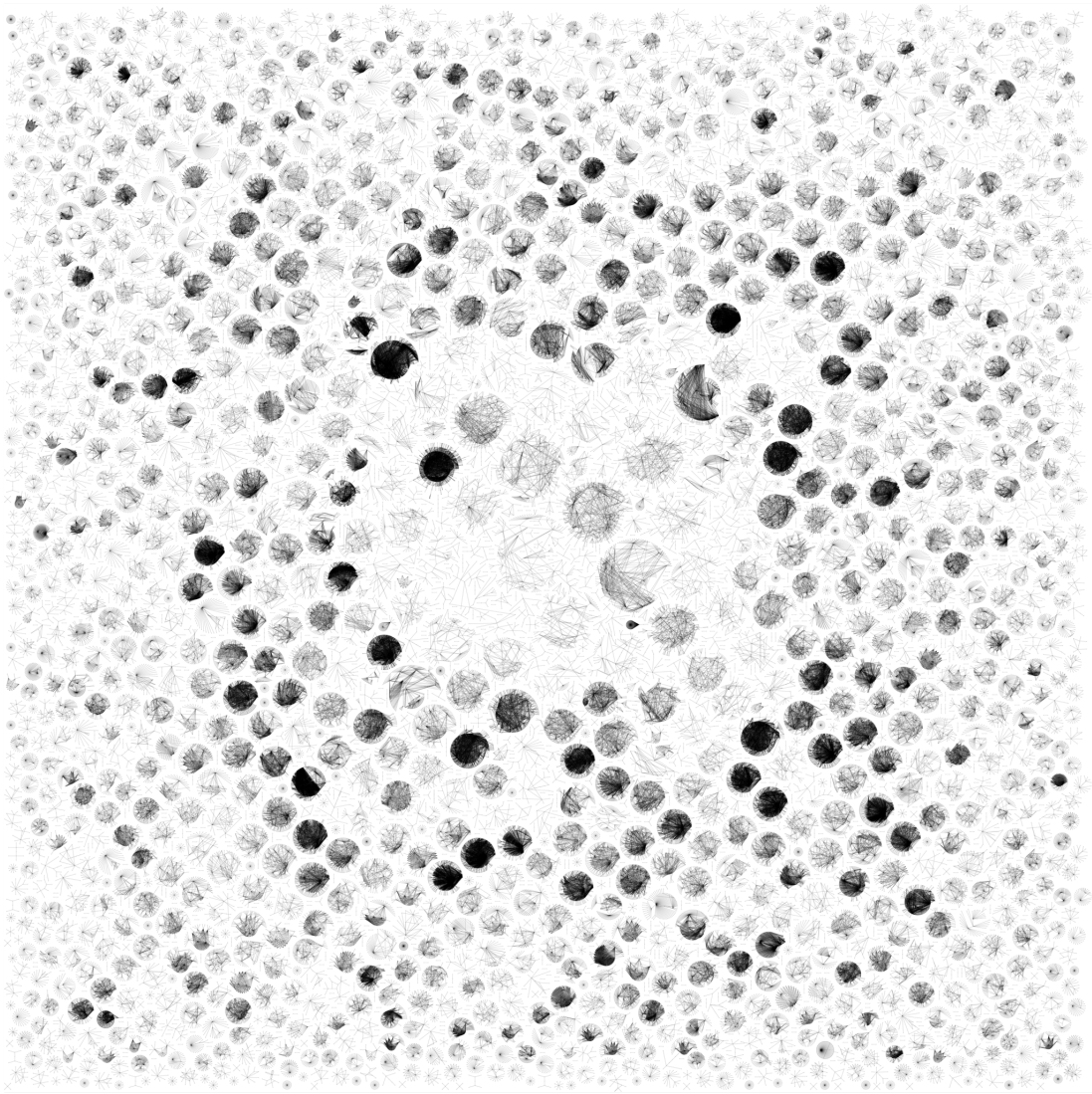


Figure 2.2: The MDG for Linux 3.2 (Figure 1.2 reproduced for convenience)

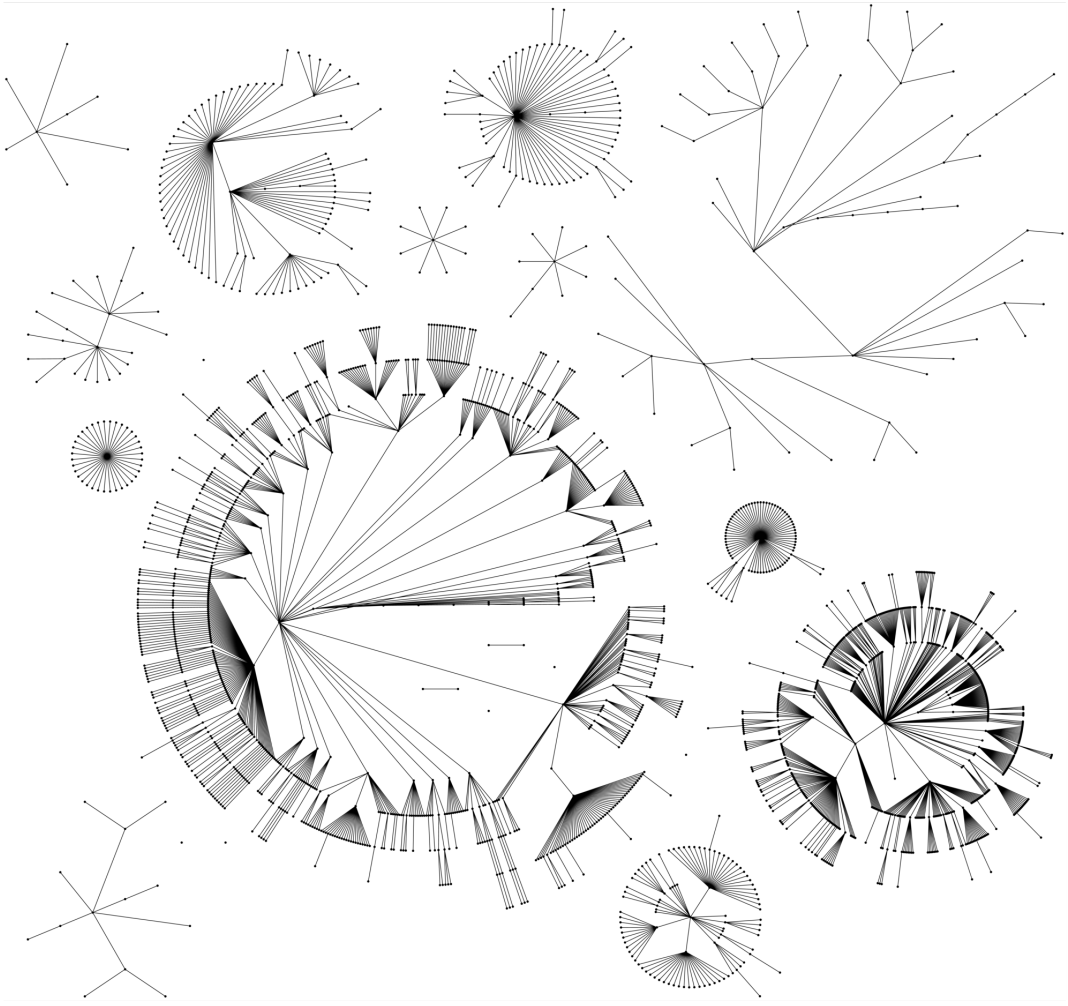


Figure 2.3: Overview of the Linux kernel's directory structure

2.1.4 The Common Problem: Automating the Abstraction Process

Both dependency graphs and state machines are graph-based representations of software, and hierarchies have been used in both cases to provide more abstraction. The equivalence of their underlying graph representation and the problem of grouping related elements makes the task of producing a hierarchy for both types of model highly similar. Table 2.1 shows the similarities between the two model types.

			Labels		
Model	Nodes	Edges	Node	Edge	Extensions
State Machine	states	transitions	Optional	Yes	Actions (Statecharts), Conditions (EFSM)
MDG	files / classes	dependencies	Yes	No	Other information (Resource Dependency Graphs), weights

Table 2.1: Similarities of the models studied in this thesis

The similarity stops at the underlying graph representation. Labels on edges or nodes are not guaranteed to be available for both diagrams. For example, state labels will not be produced automatically by grammar inference techniques although edge labels may be generated from the function names that produced the event in the state machine. Module dependency graphs, on the other hand, always have labelled nodes but do not have edge labels.

In order to unify the approaches, a solution that operates only on the available data is required. As neither edge labels or node labels are guaranteed, only the connectivity on the graph can be considered for a solution that applies to both dependency graphs and state machines.

Despite the similarity of the two models and the near-equivalence of the abstraction problem, solutions to the problem differ significantly. For state-based models, the solutions rely on the use of rule-based transformations which rewrite the graph where conditions hold. Approaches for dependency graph hierarchy generation (or "remodularisation") have applied a wider variety of techniques, including similar rule-based approaches as well as the machine learning technique of clustering.

The following section firstly describes the various goals, in the form of evaluation methods used, of approaches to the state machine hierarchy generation problem and remodularisation problem.

§ 2.2 Evaluating Hierarchies

Automated generation of hierarchies has been undertaken using a variety of methods and assessed through various criteria. Assessment typically involves the use of one or more numerical metrics which are used to make inferences about the performance of the method. The metrics used to assess the result can be broadly categorised into two groups:

- Structural quality
- Authoritativeness

Quality

The quality of a hierarchy has been assessed from a design perspective [9, 62]. These measures are highly similar to widely-used software metrics, and estimate the quality via the structure produced.

Software metrics focus on particular aspects of the system and provide a numeric overview of those aspects. In addition to performing basic measurement, metrics can be used to drive automated re-engineering processes to remedy identified problems [86].

The most simple metrics provide basic measurement of the software. For source code, for example, the number of lines of code can be used to estimate the system's size. The number of classes or files in the system (the size of V for an MDG) provides another at-a-glance view of the size of the system. Sizes of the elements in hierarchies has also been used in evaluations conducted on automated hierarchy generation approaches [9, 126].

Coupling and *cohesion* metrics are widely-used to assess the design quality of a system. These metrics represent the same coupling and cohesion discussed in Structured Design [127]. Coupling is an estimate of how inter-dependent modules are, the fewer inter-dependencies the better. Similarly, cohesion assesses how localised functionality is to the module(s) that implement it. Thus, a system with a low inter-module coupling and high intra-module cohesion is considered to be a well designed one by the principles of Structured Design.

Coupling can be measured at several levels, for example Chidamber and Kemerer [19] measured coupling between classes in their object-oriented metrics suite. Fenton and Melton [35] described how coupling at various "levels" could be calculated, ranging from no coupling, to content coupling (the former being the best and the latter the worst). Coupling at these levels was calculated based on the connectivity of modules at various levels (control flow, common variable access, variable dependencies) to estimate the overall coupling of a piece of software.

As a structural quality metric, coupling and cohesion have been used to assess the quality of MDG hierarchy generation (remodularisation) approaches [9]. The rationale behind this application of coupling and cohesion is that an ideal remodularisation approach should produce a high quality structure, judged by these metrics. Coupling and cohesion also form the basis of the quality measure used in the Bunch remodularisation tool [78], which aims to reduce the number of edges (dependencies) between different groups of files.

Cyclomatic Complexity (CC) [75] is often used to represent the complexity of software, typically for estimating maintenance costs. CC has been applied beyond its original domain of control flow graphs, being mapped to statecharts and state machines by Genero et al. [39]. CC can be calculated for any directed graph $G = (V, E)$ and is given by $|E| - |V| + 2$.

This metric for statecharts was used by Kumar [62] to evaluate a state machine hierarchy generation approach, where superstate transitions that reduced the number of edges reduced CC. Additionally, Systä et al. described a reduction in the number of edges as the goal of a hierarchy generation approach [108], although they acknowledged that a solution optimising this value may not be an optimal solution when judged by a human.

This potential mismatch between metric values and subjective opinion requires that their interpretation must be treated with caution. Anquetil and Laval [7] found that coupling and cohesion (measured at the class level) each decreased throughout several refactorings of the Eclipse IDE project. A study by Counsell et al. [26], observed a small difference between the cohesion ratings given for a piece of software between two cohorts of developers, grouped by experience. CC has also been criticised, Shepperd [102] summarises various empirical validations in which several identify it highly correlates with code size.

Authoritativeness

While metrics allow the design quality of the produced result to be estimated, they offer a one-sided view of the result. An alternative means of assessment uses a previously existing decomposition to measure how closely the algorithm reproduced the decisions made by the original developer. If the existing decomposition is assumed to be the canonical "correct" hierarchy, then the amount of similarity in the results allows the accuracy of the algorithm to be estimated. In the absence of an expert benchmark, Mitchell and Mancoridis suggested using the aggregated assignments of repeated runs of one clustering algorithm to emulate a benchmark [80].

This form of testing is widely used in clustering and has been directly applied in software clustering. The problem each of these measures solves is the inability to determine which clusters correspond between the expert hierarchy and the obtained hierarchy. For remodularisation, where the number of clusters is not fixed, it is not possible to reason about the accuracy on a cluster-by-cluster basis. For example, a cluster in the correct modularisation may be split in the remodularised solution, with elements of another cluster combined with one of the pieces. This makes judging the similarity difficult as one of the cluster fragments may be better than the other.

The solution adopted by most of these similarity measures is to treat the assignments made by the algorithm as pair-wise relations. Instead of scoring individual element assignments, the score is calculated over the set of all pairs of elements; if the pair are in the same cluster in the expert and remodularised solutions, then the algorithm has correctly assigned them. Similarly, if they are in different clusters in both solutions, the assignment was correct. Otherwise, the pair is together in one solution and apart in the other, which is an incorrect assignment.

In addition to commonly used similarity measures, some (MeCl, EdgeSim, MoJo) have been defined specifically for calculating similarity of cluster assignments. Summaries of these metrics are given in the following subsections.

Rand Index Rand Index [96] directly expresses similarity as a ratio of correct assignments to all assignments made:

$$Rand = \frac{Correct}{Correct + Incorrect}$$

This produces a number between 0 and 1 estimating how similar the two partitions are. This places an equal weight on assignments in the same cluster and those in different clusters — a pair correctly placed together counts as much as a pair placed in different clusters.

Precision and Recall Precision/Recall are commonly used in information retrieval experiments to quantify the percentage of relevant documents of those retrieved and those in the data store respectively. Unlike the Rand measure, Precision and Recall do differentiate between a correct positive and a correct negative:

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

This has been applied to remodularisation by considering pairs of elements [9], where a positive result is a pair in the same cluster and a negative result is a pair where each element is in a different cluster. Precision is then the percentage of pairs in the same cluster which are also in the same cluster in the expert modularisation. Recall is the percentage of pairs placed together which were clustered together by the algorithm.

For example, given two clusterings A and B:

$$A = \{\{1,3\},\{2,4\}\}$$
$$B = \{\{1,2,3\},\{4\}\}$$

$$Together(A) = \{\{1,3\},\{2,4\}\}$$
$$Together(B) = \{\{1,2\},\{1,3\},\{2,3\}\}$$

$$Apart(A) = \{\{1,2\},\{1,4\},\{3,2\},\{3,4\}\}$$
$$Apart(B) = \{\{1,4\},\{2,4\},\{3,4\}\}$$

The precision and recall of the clustering A for the expert modularisation B are:

$$Precision = \frac{|Together(A) \cap Together(B)|}{|Together(A)|} = \frac{|\{1,3\}|}{|\{\{1,3\},\{2,4\}\}} = \frac{1}{2}$$

$$Recall = \frac{|Together(A) \cap Together(B)|}{|Together(B)|} = \frac{|\{1,3\}|}{|\{\{1,3\},\{1,2\},\{2,3\}\}} = \frac{1}{3}$$

Precision and recall model contrasting requirements — in the extreme cases, where the algorithm clusters very few elements together, but does this correctly, the precision will be high but the recall low. In the other extreme case, where the algorithm clusters everything together, the recall will be high but the precision very low. Anquetil et al. performed experiments using a variety of hierarchical clustering algorithms and documented this behaviour. When the cut height was increased, producing larger and more general clusters, the precision tapered off, while recall increased [9], which makes interpretation of these measures difficult.

MoJo MoJo aims to estimate the cost of transforming one cluster into another in terms of the number of required modifications [114].

MoJo calculates the number of *mo*ve or *jo*in operations (hence the name). Move operations consist of migrating one element from one cluster to another, which need not necessarily exist, it is possible to move an element to a new cluster. Join operations take two clusters and merge them together.

Each operation is assigned a weight of 1, the measure is then the lowest number of these operations required to transform a clustering A into B. The MoJo distance is computed using a heuristic which aims to grow clusters around the largest fragments of the authoritative clusters that appear in the evaluated clustering.

MeCl MeCl [79] measures the distance between two partitions of a graph. Given two clusterings A and B it divides clusters in A by intersecting each cluster with each in B. These "chunks" are then merged back together to produce B. The weight of inter-edges created by the merging of these blocks is computed and expressed as a fraction of the total weight of all edges. This forms an estimate of how much restructuring was required, Mitchell normalises this by subtracting it from 1 [78].

MeCl favours partitionings which "form cohesive sub-clusters" with respect to the expert partitioning. The measure does not penalise production of more specific clustering than the expert decomposition, however. If the algorithm produces a clustering which is equivalent to a subdivided version of the expert decomposition B, its score will be 100% as no new inter-edges will be produced when its fragments are merged to form B.

EdgeSim EdgeSim [79] operates on edges (in the form of pairs of nodes) in the graph. Unlike precision and recall, EdgeSim includes both intra and inter-edges. For the same two clusterings A and B, assuming they are fully connected (each pair has an edge

between them) the EdgeSim value is defined as:

$$EdgeSim(A,B) = \frac{weight((Together(A) \cap Together(B)) \cup (Apart(A) \cap Apart(B)))}{weight(Together(A) \cup Apart(B))}$$

$$\begin{aligned} EdgeSim(A,B) &= \frac{weight(\{\{1,3\}\} \cup \{\{1,4\},\{3,4\}\})}{weight(\{\{1,3\},\{2,4\},\{1,2\},\{1,4\},\{3,2\},\{3,4\}\})} \\ &= \frac{3}{6} \\ &= 50\% \end{aligned}$$

The weight function returns 1 for all input unless the MDG is weighted. Weighted MDGs can use the number of includes between two files as an estimate of the weight.

EdgeSim differs from the other metrics in that it operates on edges rather than all pairs, and so evaluates the remodularisation algorithm on its performance of categorising edges. This does however have an effect that two completely unconnected components may be clustered together and still have a high EdgeSim score.

In an analysis of precision, recall, EdgeSim, MeCl and MoJo on 100 clustered versions of 10 case studies, Mitchell [78] noted that increases in one metric correlated to increases in others for the same pair of clusterings. For one case study, MeCl was found to have the lowest spread and standard deviation while MoJo varied the most.

§ 2.3 Metaheuristic Algorithms

Metaheuristic algorithms are used in situations where deterministic, exact solutions do not exist or are too computationally expensive. They are particularly suited to problems for which the following is true:

- generating candidate solutions is cheap; and
- quality of candidate solutions can be assessed.

These algorithms treat the problem as a search; the goal is to explore the space of all possible solutions to locate the optimum solution. Various approaches to the problem exist, all of which perform the same exploration of the space of all solutions, having different performance characteristics. This section introduces several of these algorithms.

The quality of candidate solutions is determined by an "objective function" which the algorithm uses to prioritise its exploration of the search space. Metaheuristic algorithms iteratively explore the search space until a termination condition is met, which may be determined by the amount of computation used, the state of the search (if no improvement is observed after a specified number of iterations) or user intervention.

Metaheuristic algorithms have been applied widely to clustering and remodularisation problems. This section describes the basic principles of operation of several relevant metaheuristic algorithms, as well as genetic programming (GP). Although GP is not applied in the surveyed literature, GP is used extensively in the hierarchical remodularisation work described in Chapter 4.

2.3.1 Hill Climbing

Algorithm 1 Hill climbing algorithm

```
i ← new Individual
repeat
    for all n ∈ neighbours(i) do
        if fitness(i) < fitness(n) then
            i ← n
        end if
    end for
until termination_condition
```

Hill climbing algorithms traverse the search space by considering solutions adjacent to a start point. Algorithm 1 describes the operation of a hill climbing algorithm. The starting position is selected at random and its adjacent neighbours are evaluated for increases in fitness. The termination condition is as described in the previous subsection. The selection of the new individual if there is more than one which has a higher fitness value than the current individual dictates the type of the algorithm. In a "nearest ascent" hill climber, the first neighbour with an increased fitness is chosen. This contrasts to "steepest ascent" algorithms which evaluate all neighbours and select the individual with the highest fitness.

2.3.2 Random Mutation Hill Climbing (or 1+1 EA)

Random Mutation Hill Climbers (or 1+1 EAs) can be considered to be the "most simple variant of an evolutionary algorithm" [32]. A single individual is repeatedly mutated and succeeded by the descendant mutant if the mutant has a higher fitness. The process repeats until a termination condition is reached [82, 32]. Algorithm 2 describes the RMHC implementation described by Mitchell et al. [82].

Algorithm 2 Random Mutation Hill Climber

```
i ← new Individual
repeat
    m ← mutate(i)
    if fitness(i) < fitness(m) then
        i ← m
    end if
until termination_condition
```

2.3.3 Genetic Algorithms

Genetic algorithms (GAs) operate over a number of solutions in a "population". At each iteration, the population is subjected to one or more "operators", producing a new population. This process repeats until a termination condition is reached, with

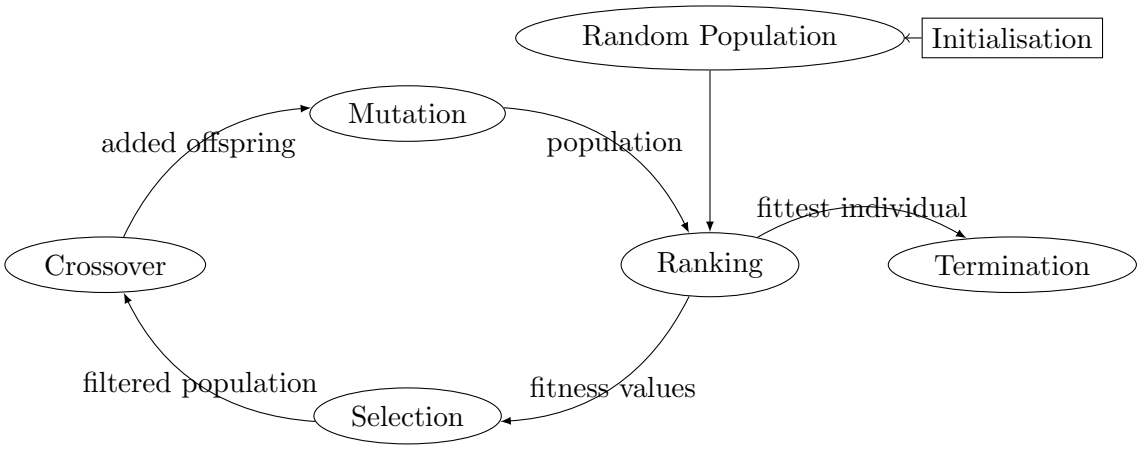


Figure 2.4: The process of a genetic algorithm

the fittest individual of the final population selected [51]. This process models a process of Darwinian evolution, where the population adapts and evolves to increase the fitness of the population over time.

The genetic operators modify or produce "chromosomes", each of which represents a candidate solution. Each chromosome is comprised of one or more genes. Chromosomes typically require decoding before the fitness function can evaluate them. The representation which encodes genes can vary between implementation. One such representation is the bit vector, which could be used to represent an integer to be optimised by the search for example. The JGAP [76] package provides a framework for the implementation of genetic algorithms in the Java programming language.

Figure 2.4 shows the process of an example GA, showing the various operators which may be used during a search. A population of random solutions is first produced which then enter an initial ranking step, before going through the cycle depicted in the figure. The order in which operators are applied can vary.

Each operator has one or parameters which affects the performance of the overall algorithm. These are in addition to the global parameters of the initial population size and stopping condition. The purpose and configurable behaviour of each operator is summarised below:

Selection The selection operator filters the population. In Holland [51, p. 92]'s original description of genetic algorithms, the selection operator is applied to determine individuals to which the crossover operator will be applied. Individuals are probabilistically selected based on their fitness. One example selection method is to apply weights to each individual according to its fitness, then use a random number to select an individual from the set. Individuals with higher weights are selected first in this "roulette wheel" selector, but less fit individuals still have a (diminished but non-zero) chance of reproducing.

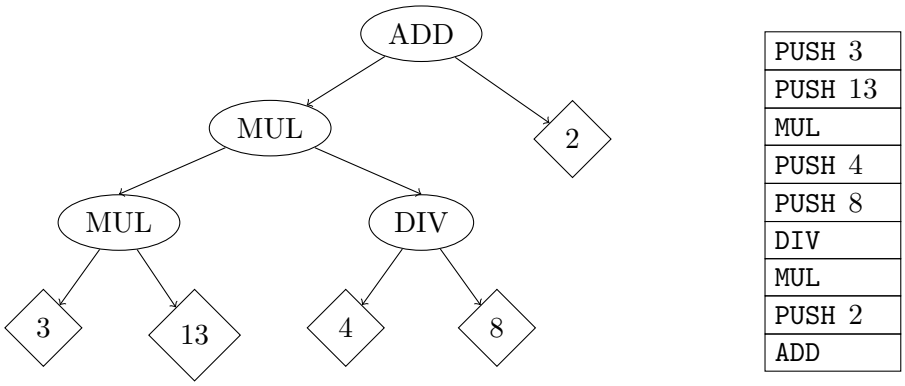


Figure 2.5: A tree-based GP genotype corresponding stack-based GP representation

Crossover The crossover operator combines individuals to produce new offspring. For chromosomes that are lists of genes, a simple crossover operator may take half of each parent and splice them to produce new individuals. More sophisticated operators exist, such as those in genetic programming where the chromosome is a tree [60].

Mutation The mutation operator introduces a random change to a chromosome. In the bit-vector example, mutation could consist of flipping a random bit. Mutation can be controlled by specifying the likelihood of a mutation.

The large number of parameters makes tuning these values essential, as well as careful consideration of the types of operator used. For example, if a selection operator routinely selects only the fittest individuals there is a chance that the algorithm may become stale at a local optimum as each individual closely resembles each other. The inverse scenario, with an excessive mutation probability, the algorithm is reduced to a random search; the selection and crossover operators are useless if fitness values randomly change for all individuals between generations.

2.3.4 Genetic Programming

Genetic Programming applies genetic algorithms to the program synthesis problem [60]. In GP, the individuals in the population are programs, which undergo the same process of selection, crossover and mutation. The fitness function is specific to the problem environment, for example, Koza [60] suggests that it may a reduction in the number of errors, or correctly classified patterns in a pattern recognition scenario or other non-functional features such as memory usage or computation time.

Koza [60] showed how hierarchical tree structures could be evolved in a genetic algorithm by defining specific mutation and crossover operators for trees. These trees are used to represent LISP style S-Expressions, encoding the sequence of operators and their operands which comprise the program.

Linear genetic programming [17] (LGP) achieves the same result as genetic programming, except instead of representing programs with trees, the chromosome encodes a sequence of instructions from an imperative programming language. A stack-based linear genetic programming technique also exists, where the individuals are evaluated by executing them in a virtual machine [91]. The advantage of these linear approaches are simplified implementation: existing list-based crossover operators can still be used. Figure 2.5 shows a tree-based genetic program and the equivalent stack-based linear program.

The linear approach also has applications where the grammar for a linear encoding of individuals is less complex than a full tree-based grammar. For example, the Java programming language has a complex grammar, but the bytecode produced by its compiler has a significantly simpler structure. This difference has been exploited in the use of genetic programming to produce general-purpose Java programs [88].

Gene Expression Programming (GEP) [36] is a similar technique to GP. It uses an altered representation to encode trees. The grammar for the tree varies depending on the problem. For example, in an application to the symbolic regression problem, GEP was used to evolve mathematical expressions that best fitted a mathematical expression to sample values from a target formula [36].

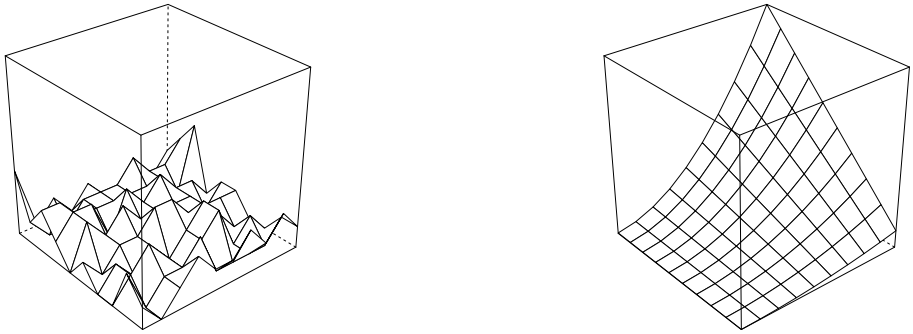
In GEP, the genotype (the gene representation) and the phenotype (the individual) are separate. GEP uses a single gene to represent the hierarchy as an expression that describes the tree. The genes are fixed-length lists of terminals and functions, which form "K-expressions" and are parsed to produce a tree. A differentiating factor between GEP and genetic programming is the notion of the "open reading frame" of the gene. Not all of an expression needs to be parsed; if a complete tree is produced before the end of the expression is met, the remainder is ignored. This is advantageous as it allows genetic operators to remain as simple implementations, rather than specific to the representation (such as tree crossover for tree-based GP).

2.3.5 Performance of Metaheuristic Algorithms

Search based algorithms are often deployed when it is infeasible to evaluate every possible solution or the complexity of algorithms guaranteed to converge on the best solution is too high. They are especially applicable where a good sub-optimal result would still be an acceptable solution to the problem

Search-based algorithms are not guaranteed to converge on an optimal solution and one of the main issues when using them is the handling of scenarios where the algorithm does not improve the solution after a number of iterations even when the current solution is sub-optimal.

This issue of the algorithm getting "trapped" in local optima can be attributed to two factors, the fitness function and the search algorithm itself. The fitness function dictates the "landscape" of the search space. When the landscape is a gentle curve to the global maximum value (the best solution) the algorithm is likely to terminate. When the landscape is uneven, the algorithm may get stuck on a peak, as neighbouring



(a) An example fitness landscape likely to cause issues with local optima

(b) A fitness landscape likely to result in convergence on the global optimum

Figure 2.6: Examples of fitness landscapes.

solutions have lower fitness values. Figure 2.6 shows examples of a good and bad fitness landscape. In some cases the fitness function can modified to better guide the algorithm, whereas in others the parameters of the search algorithm (or the algorithm itself) can be changed to make it less susceptible to getting trapped in local optima.

GAs are often considered to be more likely to escape local optima as there are multiple individuals (in most cases), which are less likely to each get stuck at the same point. In addition, the crossover and mutation operators reduce the likelihood each solution will remain in a small subset of the search space, particularly mutation as it gives the algorithm a chance to explore novel solutions which are not reachable through recombination of existing individuals.

§ 2.4 Automatic Hierarchy Construction for State Diagrams

Generating composite superstates for state machines requires determining which of the states should be placed together and which should be placed apart. This has been achieved using rule-based approaches, operating on the connectivity of the states [108, 23, 62]. While the semantics of statecharts and similar extensions to state-based models provide other means of abstraction, such as guards and concurrent states, most work focuses on composite state generation.

Systä et al. [108] described a set of transformations that can be applied to flat UML state diagrams to produce hierarchical versions which provide more abstraction. The transformations described handle two categories, process simplification and hierarchical abstraction.

The first process is a set of transformations which are applied wherever a predicate holds. These transformations specify sequences which can be merged, for instance, if a state only has one transition to another, with an action on it, that state can be replaced by the subsequent state with the action specified on entry. These transformations are specific to state diagrams which include "events", such as UML state diagrams. The type of transformation made is akin to Walkinshaw et al. [122]'s trace abstractions; a group of states is rewritten as a single state, although they do not produce readable names and depend on extra information such as actions, and trace abstractions are written over transition labels rather than states.

The second transformation described by Systä introduces composite superstates to the diagram. The algorithm is a search of all subsets which share a common transition to another state, stopping when no such subset can be found. Each matching subset is transformed into a superstate, with the largest subset matching the group being transformed first.

The theoretically optimum solution for this problem is defined as the assignment of groups which produces the smallest number of transitions in the diagram [108]. Producing this solution requires the set of all subsets to be tested and the subsequent remaining set of possible subsets required for each subset, and so on until no further partitions can be produced. This problem is NP-complete [108], and is one of the motivations for Mancoridis et al.'s use of a metaheurstic algorithm to the remodularisation problem [71]. Systä et al. also acknowledge that the theoretically optimum solution, whilst reducing the number of edges, may be sub-optimal from a human perspective.

Chu et al. [23] describe a similar technique to Systä, searching the set of states of a non-deterministic state machine and producing composite states wherever a common transition exists from each state in a subset of all states.

An extension to Systä et al.'s approach is proposed by Kumar, where concurrent states are produced by computing a decomposition of the products of states. This approach was evaluated in the same manner, although indirectly by calculating the structural cyclomatic complexity [39] of the resultant state machine [62]. The same criticism made by Systä et al. applies to this use of CC; a reduction in CC may not correspond to a meaningful partitioning of the diagram.

In all these cases, the approach depends on an expensive search to locate candidate superstates. This, along with the uncertainty regarding the correspondence of CC and understandability makes the use of these approaches to large examples prohibitively expensive. Thus, the remaining open challenges in the area of hierarchy construction for state diagrams are:

- Scalability; and
- Quality of the results.

2.4.1 Business Process Mining

Business processes are state-based models represented by a Petri net and can be mined from logged transaction data [2, 117] in a similar way to state machines. These diagrams model the sequence of a business process as a composition of several tasks, which form nodes in a Petri net. Tasks may be grouped into sub-processes. As with statecharts, it is possible for multiple tasks to be executed in parallel. Mined diagrams are likely to include verbose details which may not be necessary and as a result work exists that aims to simplify process diagrams produced from log data [30, 73]. This problem is highly similar to the state machine hierarchy generation problem and is motivated by similar circumstances.

The simplification of business processes reverse engineered from web site log data has been achieved using a variety of clustering techniques, including the rule-based matching of patterns such loops, as well as measures based on the pages which generated the traces, the dependencies between tasks and the text which generated the traces [30].

Marchetto et al. [73] proposed an alternative approach to the problem, where tasks are deleted from the process to reduce the number of tasks, and therefore the complexity. The reduction in complexity is balanced against the number of traces through the process which are made by the deletion. Thus, the algorithm aims to remove the least important tasks. These transformations resemble Systä et al.'s approach to reducing the complexity of state machines through merging of nodes according to rules [108].

Work in chapter 3 investigates the use of one of the remodularisation algorithms applied by DiFrancescomarino et al. [30], Bunch for composite state inference for state machines. Bunch is discussed in detail in Section 2.5.

§ 2.5 Remodularisation

The hierarchy generation problem applied to module dependency graphs is known as remodularisation. The objective of a remodularisation process is to produce a new set of groups for the files in an MDG (its modularisation). A large body of remodularisation work uses clustering techniques, which ignore the previous modularisation of the system and focus on imposing an potentially completely novel modularisation for the system. Tzerpos and Holt [112] provide a motivating scenario for remodularisation, wherein subsystems were reverse engineered for a large legacy system with little documentation, using a combination of facts extracted from the code and developer knowledge. This construction of alternate views and introduction of abstraction is typical of a reverse engineering scenario as described by Chikofsky and Cross [20].

An early survey of remodularisation approaches using clustering was produced by Wiggerts [125], and Anquetil and Lethbridge provide a similar survey of the application of clustering to the problem [9]. A survey by Shtern and Tzerpos discusses more recent developments in the remodularisation field [105]. Some of the following methods use search-based approaches; a survey by Räihä [95] summarises their application to software design problems, including remodularisation.

This section describes and discusses key software remodularisation algorithms, beginning with a description of the ACDC pattern based method. The remainder of the algorithms use variants of clustering, which is described before these methods are discussed in the following sections.

2.5.1 ACDC: Pattern-based Remodularisation of MDGs

The ACDC remodularisation algorithm is designed for comprehension [115] and consists of two stages: skeleton construction and orphan adoption. Similar to approaches for state machine hierarchy generation, ACDC's skeleton generation step identifies parts of the dependency graph that match patterns.

ACDC builds up on the skeleton by adding the remaining elements, based on the dependencies between them. The cluster with the highest number of dependencies on each element "adopts" it [113]. This algorithm produces a full hierarchy, the sizes of the subsystems of which can be controlled by user-specified limits.

MoJo similarities (in terms of proximity to the expert decomposition) produced by ACDC for two case studies were 64.2% for TOBEY and 55.7% for Linux respectively [115]. Analysis of the stability, the result of clustering randomly perturbed versions of the case studies [116], found that ACDC produced similar MoJo values for similar case studies [116] in 81.3% and 69.4% of cases respectively

ACDC's reliance on patterns makes it inapplicable without modification, however it only operates on the dependency graph. The elements contained in the skeleton are used to produce names for the subsystems identified by the algorithm, however reverse engineered state machines are unlikely to have this information.

The majority of other remodularisation approaches apply a clustering approach to the problem. The remainder of this section details these clustering approaches, beginning with a general description of clustering algorithms, focusing on hierarchical algorithms in particular.

2.5.2 Clustering

Clustering is a machine learning task wherein groups are inferred for a data set automatically that has been widely applied to the remodularisation problem. Clustering algorithms aim to group elements of the data set together based on their apparent similarity, as expressed by the "features" they exhibit. This section briefly outlines the fundamentals of clustering; Berkhin's survey contains a comprehensive description of various algorithms as well as various distance metrics [14].

Figure 2.7 shows an example of the use of k-means clustering [58] on a set of data with two features. In this instance, the number of clusters k was set to 3. This shows the operation of a clustering algorithm in general; the objective is to select arrangements that maximise the similarity between elements within the identified groups.

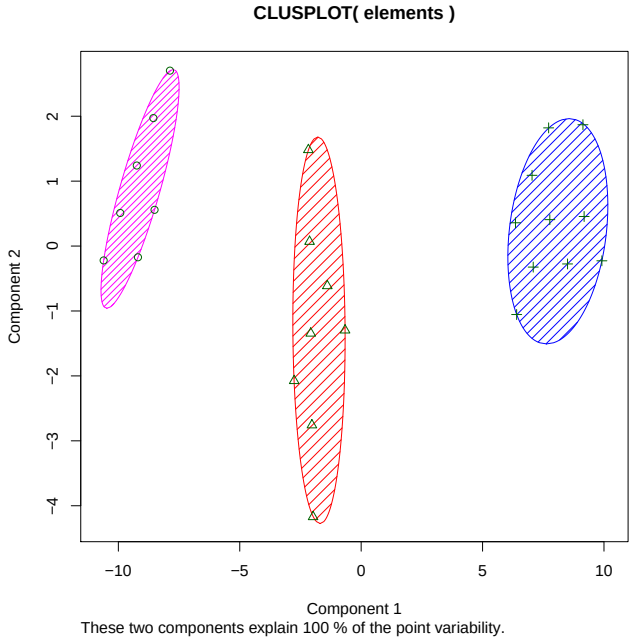


Figure 2.7: an example of the use of k-means with a 2D data set, $k = 3$

Clustering algorithms commonly compute groups based on similarity measures, which are used to map points in the feature space to distances. Pair-wise similarity values can then be used to guide the group production process for the data set. The mapping to similarity values depends on the type of data used. Binary features, for example, can be treated as coordinates directly, with the Euclidean distance used to estimate the similarity of the data [118].

Algorithms that use similarity measures can be categorised based on the approach of assigning groups used. Hierarchical algorithms cluster the data one pair at a time, producing a full hierarchy for the data set [58]. Other types exist, such as density based methods and partitioning methods, although only hierarchical algorithms have been applied to the remodularisation problem, so this section focuses solely on these approaches.

Hierarchical algorithms produce a full tree for the data, without requiring selection of the number of clusters prior to running the algorithm. These methods may be either agglomerative or divisive, where the clusters are iteratively constructed or divided respectively [58].

The tree for each type of algorithm is constructed as it makes cuts (or joins). The order of the operations determines the depth of the cluster in the tree. Hierarchical algorithms perform their merge (or cut) operations in an order dictated by the similarity metric; the most (or least) similar elements will be joined (or cut) first, and thus appear higher (lower) in the tree.

These algorithms also assign heights to the branches in the tree, derived from the distance. A "cut" can be made after the clustering to produce a single partitioning, the height of which determines the size of the clusters (a higher cut corresponds to fewer, larger clusters). The selection of the cut-point is specific to the scenario and the result (much like k in the previously discussed partitioning approaches).

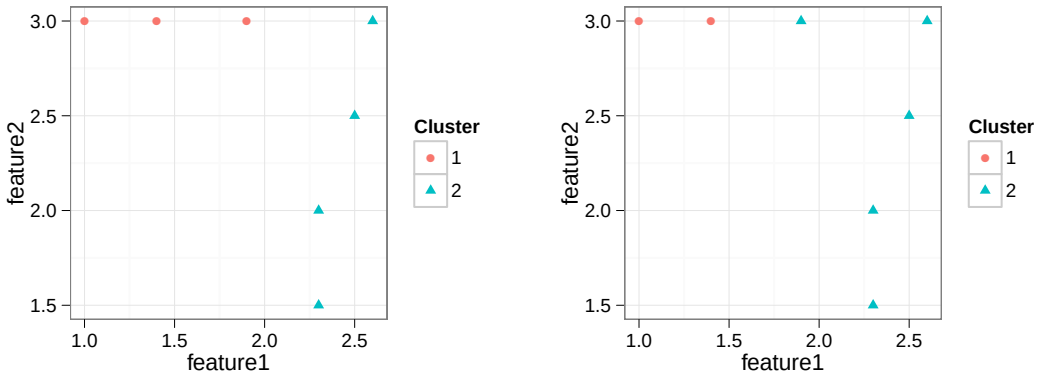


Figure 2.8: Cluster assignments with **(a)**: single, **(b)** complete linkage [125]

Hierarchical algorithms differ by the method used to estimate the distance between clusters. This is referred to as "linkage". Linkage metrics typically determine cluster similarity between one or more of the pairs of elements in each cluster. The minimum, maximum and average are known as single, complete and average linkage respectively [14]. Figure 2.8 illustrates the impact of the choice of linkage on the result. The point at $(1.9, 3)$ is clustered differently with each linkage type.

Evolutionary Clustering

As alternatives to partitioning and hierarchical methods, evolutionary approaches have been applied to the clustering problem. The distance measure can be encoded in the fitness function, the genotype is the assignments of modules to clusters. This section introduces the basic methodology of evolutionary clustering; a survey by Hruschka et al. [52] discusses the various representations and algorithms described in the literature.

An evolutionary approach has been used for single level partitioning [61] of objects, where the phenotype is a linear vector which encodes a cluster ID for each data element. Figure 2.9 shows an example. In this case, there are 3 elements to be clustered, the first two of which are assigned to cluster number 1 and the third assigned to cluster 3.

1	1	3

Figure 2.9: A flat label chromosome

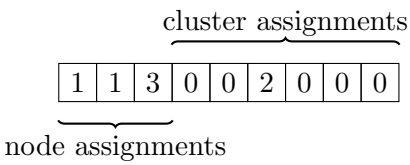


Figure 2.10: A hierarchical label gene

Hierarchical clustering has also been achieved using a genetic algorithm and a similar representation [22, 21]. The flat chromosome is extended to include a table structure that maps clusters to parents, forming a tree. This tree can be either a binary classification tree [22] or a full hierarchical dendrogram [21].

Figure 2.10 depicts an example of this hierarchical label representation. As random mutation and crossover can produce invalid solutions, Chis [21] defined specific operators to prevent them occurring. Crossover is defined as a swapping of hierarchy assignments between parents. Mutation is applied to the node assignments to constrain node assignments to terminal nodes in the hierarchy [21].

De Falco et al. [34] also describe an application of genetic programming to the clustering problem by evolving predicates, generated from a grammar. Their approach allows the number of clusters to be determined by the algorithm, differentiating it from partitioning algorithms such as k-means which require a-priori selection of the number of clusters. The approach produces a single level of clustering, in contrast to Chis' hierarchical approach. In the fitness function, the aggregated internal similarity and weighted external dissimilarity are used. The external dissimilarity is weighted to allow the user to select the appropriate value depending on the scenario. Similarity values correspond to distance measures discussed earlier in Section 2.5.2.

Constrained Clustering Algorithms

Wagstaff et al. introduced a constrained variant of the k-means clustering algorithm [120], following with a constrained variant [119] of the COBWEB [37] conceptual clustering algorithm. Assessment of each of these algorithms showed they outperformed their un-constrained counterparts with small numbers of constraints and, in the case of COP-KMEANS, performance of the combined constrained clustering exceeded the sum of the performance of the individual clustering or constrained approaches. This demonstrates the practicality of using additional domain knowledge to boost the performance of clustering algorithms.

2.5.3 Software Clustering

Applications of clustering algorithms to the remodularisation problem are widely discussed [6, 8, 9, 84, 100, 118], with the majority focusing on the use of hierarchical clustering algorithms. Wiggerts [125] provides an overview of the applicability of clustering to the remodularisation problem and describes the different types of algorithms,

the impact of linkage on the result and various methods of mapping features to distances. This section summarises and discusses several relevant techniques; a comprehensive survey and evaluation of hierarchical algorithms is provided by Maqbool and Babri [72].

The application of hierarchical clustering to the remodularisation problem requires the following:

Selection of the algorithm Most use hierarchical algorithms, although the linkage types differs

Features The type of data which will be used to compute similarity values

Similarity measure The function that maps the feature values of two elements to a distance value

Hutchens and Basili [53] originally described an application of hierarchical clustering to the remodularisation problem, by using several types of linkage to modularise procedures bound by the variables they used. Distance values from these "data bindings" were translated into a dissimilarity measure which aims to group procedures that use similar variables together.

Assessment of their technique used a database of issues associated with components to determine how well faults were localised to clusters. They concluded that "recomputed bindings" produced the best result, which awards high similarities to pairs with the highest proportion of common dependencies.

Arch [100] applies a similar approach using a heuristic which estimates the "design coupling" between functions and types. Design coupling is based on the dependencies of functions on types and represents the impact that changing a type would have on its dependent functions. A "design link" exists between a function and each type it uses; this link is used in the similarity measure. Arch remodularises functions into files (and files containing them into modules). Schwanke remarked [100] that single linkage produced the "most promising" results.

In addition to performing unsupervised modularisation, Arch provides supervised modes where a developer confirms merges before they are carried out. If the expert determines a merge should not be made, this negative feedback persists and the algorithm will not attempt to merge the pair again. Arch can also refine an existing modularisation. During this process an expert is consulted if merges result in changes to the previous modularisation. The developer can also provide domain knowledge through the tuning of parameters to the similarity measure.

Three architects reviewed the modularisation produced by Arch for its own code. Of 56 merge operations made, 40 were the same in the original modularisation. The panel determined that 10 of the remaining changes which moved procedures between modules were correct and 6 were incorrect [100].

The performance of different types of hierarchical clustering has also been measured for a remodularisation scenario; Anquetil and Lethbridge evaluated different types of

linkage (single, complete, weighted average, unweighted average) [9] for various features (file includes, return calls, type references and word references in identifiers — extracted using their concept identification method [8]). Each of the combinations of these facets of the clustering algorithm (feature, linkage type, feature used, similarity measure) was applied to four case studies: Linux, Mosaic, GCC and a telecommunications system.

The results were evaluated through both design quality and authoritative metrics at each cut point, using measures discussed in Section 2.2. Design quality was assessed through coupling and cohesion as well as the sizes of clusters in the result. The size measure split the clusters into three categories: the size of the largest cluster, the number of singletons (containing only one element) and the remainder not in the previous two groups. Authoritativeness was measured with precision and recall using the directory structure as the expert decomposition.

The results from this study showed a wide variance in performance for each feature between case studies. Zero values in feature vectors were found to be problematic, as they influenced the distance measure and distance measures ignoring them were found to produce better results [9], making elements that had very few features appear more similar than they actually were. Of the features used, Anquetil and Lethbridge remarked that the "words in identifiers" feature produced good results, despite it not being a structural feature of the code.

A similar application of hierarchical clustering has been applied to the object identification problem for COBOL software [118]. Variables were remodularised to produce candidate groupings which could be used as a basis for classes. The feature set was a binary vector for each variable's presence in each of a set of programs, with the hypothesis that variables are related if they each appear in the same programs. The variables were clustered using the AGNES [58, p199] hierarchical clustering algorithm and the Euclidean distance for the similarity measurement.

Due to the sparse nature of the feature set, the distance between many variables was calculated to be the same. van Deursen and Kuipers suggest that this results in the clustering algorithm resorting to non-deterministic choice when computing clusters, potentially resulting in a compounding effect on the error introduced. The use of other metrics as suggested by suggested by Wiggerts [125] did not produce an improvement, although removing programs which referenced all variables from the feature set was found to produce a better clustering of the variables.

Naseem et al. [84] surveyed various similarity measures, including those originally suggested by Wiggerts [125]. An extension of the Jaccard similarity measure was shown to produce fewer conflicts of similarity where multiple elements have the same similarity. These conflicts result in the algorithm resorting to non-deterministic choice of elements to merge, a problem identified by van Deursen and Kuipers [118]. Although this alternative measure decreased the number of arbitrary decisions, the authoritativeness values (measured using the MoJoFM metric) were found to be inferior to those produced using the original Jaccard measure.

Another issue with clustering was raised by van Deursen and Kuipers when comparing it to the use of concept analysis for object identification: when using hierarchical methods,

the developer must select the cut height to transform the dendrogram into a set of objects, which, for a large system is likely to be difficult [118]. For this reason, concept analysis was recommended for the class identification problem [118].

MULICsoft [6] adopts a similar clustering approach to the problem, however it is not based on the same hierarchical clustering method as the previously discussed methods. MULICsoft clusters a weighted module dependency graph, where weights are obtained through dynamic analysis; more calls from one file to another increase the weight. It clusters elements together based on a threshold (ϕ) applied to a similarity metric, calculated between each element to be clustered and the mode of each already-present cluster. The mode of a cluster is given by finding the most common value for each feature from the elements within it. Pairs of zero valued features are ignored in the calculation of the similarity measure, as recommended by van Deursen and Kuipers [118].

ϕ is increased if none of the elements to be clustered is assigned after an iteration. A user-specified maximum threshold parameter stops the algorithm when ϕ exceeds this limit. ϕ is used to build layers: elements clustered earlier (with a lower ϕ) are more similar and thus appear at higher levels. Singleton clusters are removed from the cluster solution at each iteration. This prevents singletons appearing high in the hierarchy, as instead they will be clustered at a lower level (when ϕ is sufficiently large).

MULICsoft was evaluated through comparison with other algorithms (ACDC, Bunch, LIMBO) using various values for the step-wise increase in ϕ, different weighting schemes for the feature matrix. This evaluation included comparison of the resultant decomposition with an existing decomposition, using the MoJo similarity metric. MULICsoft was observed to produce a superior MoJo value to the other algorithms for one case study containing 1202 files. Of four algorithms applied to this case study, its best result contained 191 clusters. ACDC produced a similar number, 205, while Bunch was found to produce the fewest clusters (21).

2.5.4 Concept Clustering

In addition to hierarchical clustering, van Deursen and Kuipers also investigated the use of concept analysis to modularise variables by programs [118]. Concept analysis places each possible grouping of elements on a concept lattice. Each of the solutions on this lattice has a common *concept*, in this context a feature (or an element's presence in a program) is a concept. The concepts form a partial ordering, from most general to least general (exhibiting the fewest concepts to the most concepts).

On comparing concept clustering to traditional clustering, van Deursen and Kuipers noted that clustering produces only one of many possible results (determined by the sequence of merges and choice of cut point) while concept analysis produces only one lattice for a given feature set. Concept analysis is also not constrained to assign each element to only one class, making it more resilient where the data contains commonly present features. This negates the necessity to remove these features, as conducted in their hierarchical clustering experiment [118].

Tonella also investigated the use of concept analysis for remodularisation [111]. In this case, functions were clustered according to their use of structured types (`structs` in C), dynamic memory heaps and global variables. This concept analysis approach was assessed using structural metrics, specifically focusing on three aspects:

- Modularisation cost;
- Encapsulation violations; and
- Modularisation quality.

Modularisation cost modelled the amount of effort required to adapt to the new modularisation. This is estimated by calculating the distance between the remodularised version and the original version. Encapsulation violation is similar to the use of a coupling metric by Anquetil and Lethbridge [9]; cases where a function used data from other modules estimate how well functions are grouped. Finally, the measure of modularisation is a count of the number of modules produced.

The concept clustering approach is able to produce a large number of candidate partitions. As a result, Tonella ran the algorithm for a fixed amount of time, selecting the best elements from those generated. The process was applied to 20 case studies; 10 open source systems and 10 industry systems. For one case study, the modularity was increased with the encapsulation violation metric remaining unchanged and the cost remaining low for some solutions, while others showed a reduction in encapsulation violations and increase in modularity at a much higher modularisation cost.

2.5.5 Graph Clustering

Graph clustering is a mapping of the clustering problem to general graphs. Algorithms such as Girvan-Newman clustering have been used in the analysis of social networks, and have also been applied to software.

Network Clustering for Visualisation The Barrio [31] clustering tool uses the Girvan-Newman graph clustering algorithm to identify modules based on the connectivity of the MDG. Girvan-Newman treats the graph as a network, and clusters graphs by iteratively removing the edge with the highest number of paths running through it. Each deletion which isolates "communities" creates clusters, which are further divided until a user-specified separation level is reached. Barrio visualises the resultant clusters using the JUNG [54] graph visualisation toolkit.

Min-Cut for Component Extraction Marx et al. demonstrate the application of the min-cut graph partitioning algorithm to software modularisation [74]. Taking a user's selected assignments into consideration, a system is bisected using an optimisation algorithm; either hill climbing, min-cut, an exhaustive search, or a combination of the three. The user can then refine this by providing more information before the algorithm is run again. A set of interfaces which "bridge" the gap between the two

components is computed from the edges cut by the bisection operation. This allows development of each component to continue in parallel, and also serves as the visualisation for the partitioning: the user is shown a UML diagram which includes the interfaces.

A small user study found the participants identified problems with their code after performing the extraction — they remark that one participant "was able to identify a component but was not completely satisfied because two obsolete classes interfered with the extraction" [74]. Additionally, they found that the task allowed the participants to reflect on the design of the system, suggesting the act of component extraction may be suitable for detection of poor design choices.

Scalability of their approach was reportedly a problem: only min-cut of the three algorithms used was feasible for a 3000 node project. The amount of data to be visualised during the process also presents a challenge which Marx et al. proposed to solve using zooming and filtering techniques.

Rigi Müller et al. implemented graph-based approaches in a semi-automatic hierarchical modularisation environment, Rigi [83]. Rigi builds hierarchies from "building blocks". These building blocks are identified by a human, but Rigi provides various tools to aid in the modularisation process. Operating on *Resource Flow Graphs* (RFGs), that generalise dependency graphs to include any type of information for which there are resource relations defined (i.e. there are producers and consumers), Rigi constructs a composition dependency graph (CDG) which represents the full hierarchy of the system where the terminal nodes are the elements of the RFG. There is no constraint on the number of clusters an element can appear in, thus CDGs are not trees, but acyclic directed graphs.

Rigi includes measures over elements of the RFG to aid in modularisation:

Interconnection Strength The number of edges between a pair of vertices in the graph

Common Clients/Suppliers The set of all vertices with edges (dependencies) to/from all vertices in a module

Centricity The sum of all weights of edges between a vertex v and any other element which isn't contained within v

Name Edge names matching a user-supplied regular expression

These all rely on the user supplying thresholds (or a regular expression). The Rigi user interface allows the user to explore different parameters and composition operations.

2.5.6 Search Based Remodularisation

Search-based methods have been widely studied in the remodularisation field. The motivation for treating the remodularisation problem as an optimisation task is due to

the ease of generating candidate solutions and the ability to produce objective values for these candidates (via metrics). The majority of the search-based approaches discussed in this subsection apply metrics based on coupling and cohesion to the problem, aiming to locate solutions with a high quality architecture (as assessed by these metrics).

Bunch

Bunch [81] is the most widely used and surveyed search-based remodularisation technique. It produces hierarchical decompositions, built up in layers, using either a hill climber, simulated annealing, or genetic algorithm. Bunch only relies on the connectivity in the MDG for the feature set, but also includes support for optional weights on the edges in the dependency graph.

Bunch takes an agglomerative approach to the problem. First, the files or classes are clustered into small modules. Subsequent searches merge the modules from the previous search to produce a layered hierarchy.

For the genetic algorithm, the label-based representation described by Krovi [61] is used, where a chromosome is a vector, where each element in the vector contains the cluster for the module at that position is used. The specifics of this representation are discussed in Section 2.5.2.

The fitness function used in Bunch aims to maximise the cohesion of clusters while minimising coupling between clusters. These objectives are encoded in the Modularisation Quality (MQ) fitness function, which is expressed as a ratio of coupling to cohesion. MQ is given by:

$$MQ = \sum_{i=1}^{k} CF_i$$

$$CF_i = \frac{2\mu_i}{2\mu_i + \sum\limits_{j=1, j\neq i}^{k} (\epsilon_{i,j} + \epsilon_{j,i})}$$

Where μ_i is the number of edges within cluster i ("intra edges"), $\epsilon_{i,j}$ is the number of edges between cluster i and cluster j ("inter edges") and k is the number of clusters.

MQ is not unlike the similarity metric used in Arch [100]; MQ seeks to maximise the proportion of edges within a cluster to those leaving or entering it. It differs from Arch in that it operates directly on the MDG, while Arch operates on the design linkage between functions and types. MQ contrasts from the distance measures typically used in hierarchical clustering, however. By treating the remodularisation as a set of feature vectors, algorithms group files by the similarities of their dependencies; for example, if two files each depend on the same set of files, they are similar. MQ differs, by aiming to cluster files as close to their dependencies as possible.

The use of a ratio makes the "cluster factor" (CF) more tolerant of inter edges for larger clusters, however the maximum value for CF is 1.0. This limit incentivises the

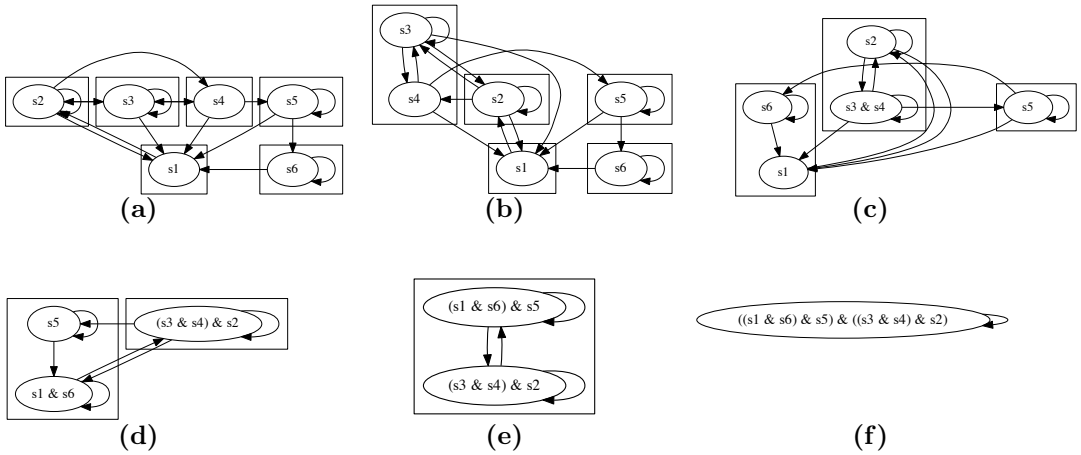


Figure 2.11: Iterative hierarchy generation in Bunch; each part is a search, from the bottom level in part **(a)** to the top level in part **(f)**

creation of more clusters, however the CF term requires the number of clusters to be balanced against the coupling between modules. As Bunch removes reflexive edges, singleton clusters are always assigned a value of 0. This ensures the algorithm will rarely produce singleton clusters.

The agglomerative process Bunch uses to produce hierarchies is portrayed in Figure 2.11. The search begins by clustering the MDG (part **(a)**). In each subsequent search (parts **(b)** – **(f)**), the clusters from the previous search, and their inter-edges, are clustered by Bunch again. Each search produces a new level in the hierarchy in a bottom-up manner. Bunch stops when the search produces a single cluster by itself (shown in part **(f)**).

Bunch also includes features to restrict the search to only components that require remodularisation. Omnipresent modules can be identified based on an above-average edge count and excluded from the search [81]. Libraries can also be detected in a similar manner: nodes in the graph which do not depend on other nodes (but other nodes depend on them). These options are user-driven in the sense that a user must judge the quality of the results of the library and omnipresent module detection processes. Support for user-driven clustering is also integrated into Bunch; a user can supply a clustering which the algorithm will start from and specify elements of the clustering to be locked.

The Bunch algorithm has been evaluated in an industrial context: Glorie et al. [40] found that, on using Bunch in a real-world scenario, although Bunch did identify some useful modularisations, the majority were poor. This impacted the utility of the modularisation provided by Bunch, rendering it unusable in this industrial setting. This contrasts with the reported agreement with Bunch-generated clusterings by developers of systems [77, 81].

Bunch has been reported to consistently produce highly similar decompositions for the same input MDG [81]. Mitchell and Mancoridis hypothesised that files that between modules may be indicative of poor design choices, an observation also made by Schwanke [100] in the investigation of "maverick" procedures identified by the Arch clustering tool.

Improvements on Bunch Performance

Harman et al. [46] demonstrated that a genetic algorithm using an alternative representation which reduces the search space by limiting the number of ways to represent a solution outperformed the label-based representation [61] used in Bunch, but performed worse than a hill climbing algorithm.

Efforts to improve on Bunch have focused mainly on the use of alternative search techniques to yield higher MQ values. Mahdavi et al. [70] used multiple hill climbs to locate better starting configurations (building blocks) for the search.

Harman et al. [47] analysed the performance of the MQ fitness function used in Bunch and compared it to the EVM fitness function. The two fitness functions were evaluated 6 software systems and 6 simulated case studies, three with several completely connected components and three with random connectivity, with an average edge density equal to that of the software systems.

Each of these case studies was clustered with an increasing amount of noise, generated by randomly modifying the MDG's adjacency matrix. The perfect MDGs were compared to their known-good decomposition, which was not available for the software systems. In place of an expert decomposition, the clustering for the MDG with no introduced noise was used. The similarity between pairs of authoritative decomposition and clustered decomposition was compared, with the results showing that EVM was more tolerant to noise.

The use of a multi-objective genetic algorithm approach has been found to produce higher MQ values than the Bunch hill climbing algorithm for the same number of fitness evaluations [94], albeit at an increased computational cost.

Search-based Refactoring

Refactoring is defined by Fowler as the process of modifying a piece of software to improve its design without altering its behaviour [38]. For example, a large class that is coupled with many other classes may be divided into several smaller classes to reduce the coupling of the design. This problem is similar to remodularisation; the objectives of increased design quality are the same. Refactoring differs from remodularisation in that more than one type of entity can be manipulated (e.g. moving a method implementation to a superclass), and that some solutions are not valid (they may fail to compile).

Seng et al. [101] applied a genetic algorithm to the problem of identifying refactorings for software. In their approach, they evolved sequences of refactoring operations, which

are descriptions of transformations to be made to the design of the system, such as moving a method from one class to another. The fitness of the results of this process were evaluated using a weighted sum of several metrics, which included coupling and cohesion [101].

Harman and Tratt expanded upon this approach with a multi-objective search [48]. Rather than applying a genetic algorithm, their approach adopted a hill climbing technique in order to identify a "Pareto-optimal" set of modularisations that best optimised the coupling between object metric and the standard deviation of methods per class. Their approach was motivated by the difficulty in selecting metrics for use in a weighted single-value measurement.

These "indirect" approaches divide the genotype and phenotype. The phenotype, the refactored system, is produced as a result of applying the genotype, a sequence of refactoring operations, to the original system. This indirect approach is beneficial as it allows a global search to be applied as mutation and crossover cannot produce individuals that result in invalid solutions [48]. The equivalent direct approach, where the genotype is a model of the system mandates that the crossover and mutation operators preserve the semantics of the program.

Search-based remodularisation using the Minimum Description Length Principle

Lutz [68] demonstrated an application of the minimum description length (MDL) principle [97] to the remodularisation problem. A fitness function which uses the length of the sequence of bits required to encode the tree estimates the quality of the candidate solution. This fitness function was used with a genetic algorithm and a tree-based representation to produce hierarchical decompositions for a software system.

Evaluation of this technique was conducted for a small operating system (MiniTunis) and compared with the result from Bunch [69]. Lutz remarks that the MDL approach produced good low-level clusters but the hierarchy contained excessive nesting of modules. Although Bunch was found to produce a less similar result, Lutz makes the observation that the MDL complexity of the result from Bunch was lower than the original decomposition, suggesting MQ and the MDL fitness function prioritise similar features.

Software Renovation Framework

Di Penta et al. use remodularisation as a component in their "Software Renovation Framework" (SRF) [90]. The SRF performs several stages of analysis and refactorings to remove unused objects, detect and merge clones and remodularise libraries so they can potentially be transformed into dynamic libraries. The clustering stage is used to transform libraries into smaller libraries. The AGNES clustering algorithm is first used to produce a sub-optimal arrangement. A genetic algorithm is then used to refine the result so it optimises a weighted sum of the fraction of number of functions used from

the library and the size of the new libraries. The refinement step is repeated until a developer indicates the refactoring is satisfactory.

Interactive Multi-Objective Genetic Algorithm

Work using a multi-objective approach has also leveraged interactive input to further improve the clustering process [12]. This approach is similar to the user driven merging in the Arch tool [100], however while Arch confirms each merge of elements with a user, the IGA approach solicits information in the form of constraints for a subset of the assignments, which are incorporated into the fitness function.

Bavota et al. outline two alternative implementations, which differ in the selection of the elements to solicit feedback for. The R-IGA approach makes a random selection, while the IC-IGA algorithm solicits feedback for isolated clusters (or the smallest if there are no isolated clusters). In both variants, the user gives feedback in the form of assignments the elements to other clusters. Each of the variants parameterise the amount of information solicited at each feedback step as well as the number of feedback steps before finishing.

In the single objective implementation (IGA), the feedback is solicited for elements selected from the best solution (highest fitness), based on either random (R-IGA) or the size of the clusters (IC-IGA). The user-supplied information is applied as constraints to the solution, and the modified version is used to seed a new population and the process repeats. The fitness function incorporates the constraints by dividing MQ by a weighted sum of all violated constraints:

$$F(s) = \frac{MQ(s)}{1 + k.\sum_{i=1}^{m} vcs_{i,s}}$$

Where $MQ(s)$ is the MQ value of the solution, as defined by Mitchell and Mancoridis [81], and $vcs_{i,s}$ is 1 if constraint i is violated in the solution s. The k term allows the weight applied to constraint violation to be adjusted.

The multi-objective IGA implementation (IMGA) employs the NSGA-II multi-objective genetic algorithm [29]. The IMGA algorithm has two variants, similar to the single objective algorithm: evolve a population of solutions and stop after a pre-determined number of generations, select the solution with the highest MQ value, then select the element(s) for which feedback will be solicited, repair the solution, generate a new population from it and repeat the process.

Bavota et al. assess the performance of the interactive algorithms by comparing them against their non-interactive equivalents. This comparison uses the MoJoFM [124] distance measure against a known authoritative decomposition for two case studies. Instead of using humans for the feedback, the authoritative decompositions were used. The amount of feedback was set to 5 interaction steps (where the evolution is halted and corrections solicited), each consisting of 3 pieces of feedback (so the interactive algorithms use a total of 15 corrective interactions).

The interactive variants of both types of algorithm found higher quality solutions than the non-interactive versions. Additionally, the algorithms limited to a smaller number of clusters (half the number of files) consistently performed worse than the less restricted (number of clusters $\leq$ number of files) runs. Despite this, the more restricted searches produced solutions with fewer singleton clusters, Bavota et al. cite an instance where the less restricted GA search produces on average (mean) 149 clusters, on average (mean) 72 of which were singletons.

The MQ values obtained by the remodularisation algorithms were higher than the MQ value of the original expert decompositions in some cases. This corroborates Praditwong et al.'s remarks on their multi-objective remodularisation algorithm:

> *It is also important to note that this work neither demonstrates nor implies any necessary association between quality of systems and the modularisation produced by the approach used in this paper [94].*

This highlights both the importance of validating fitness functions and also the problem automated (or semi-automated) approaches face; subjectivity of the developer will remain a problem unless it can be incorporated into the process (or the resultant solution).

This section has shown that a large body of work exists on the remodularisation subject. The majority of these approaches apply a flexible clustering algorithm, and operate on the limited feature set of the dependency graph. In some cases, additional information is used, either from the system, such as Rigi's use of additional types of information, or a domain expert, in the case of Arch [100] and the interactive genetic algorithm used by Bavota et al. [12].

The remainder of this chapter evaluates the work discussed with regard to the application to the state machine hierarchy generation problem. Focus is placed on the previously described flexible approaches that do not depend on information specific to MDGs.

2.5.7 Performance of Remodularisation Algorithms

The objective quantification of the performance of clustering algorithms is made difficult by the differences in criteria and in the data that algorithms use to produce their results.

Wu et al. [126] performed a comparison of 6 clustering algorithms over a set of 70 – 73 monthly revisions of 5 systems. Each revision of each case study was clustered by each algorithm and the results were assessed in three ways:

Stability How similar the modularisations were between revisions

Authoritativeness How similar the modularisations were compared to the revision's modularisation (file system structure)

Cluster size The range of cluster sizes produced for each revision

MoJo similarity values were calculated for each consecutive pair of decompositions for each case study and for each algorithm. On comparing these values to the growth of the system, Bunch was found to be the least stable, while ACDC and both single linkage hierarchical clustering algorithms were found to be the most stable.

Authoritativeness was compared using the filesystem structure, which was modified by moving all header files (`.h`) to the directory containing their corresponding implementation (`.c` files). MoJo similarities were calculated for each clustered revision between the corresponding authoritative decomposition.

On comparing authoritativeness, Wu et al. [126] remarked that none of the surveyed algorithms produced modularisations similar enough to the authoritative decompositions. However, the complete linkage algorithm with a 0.9 cut height produced the highest overall authoritative similarity values.

Sizes of clusters produced were measured for each of the algorithms, then subjected to a "NED (non-extreme distribution)" measure. The percentage of elements in non-extreme clusters, defined as those with more than 5 and less than 100 elements of the total number of elements were calculated. A score of 100% constitutes a modularisation with no extreme modules. Bunch was found to produce the fewest number of extreme clusters [126].

Other studies have focused on single level performance; a comparison of Bunch with the ACDC clustering algorithm on simulated systems found that Bunch produced the most authoritative decompositions for two of the three systems tested [104]. However, ACDC was also found to produce more authoritative hierarchies than Bunch [103] although Bunch produced partitionings which cut a higher number of edges correctly than those produced by ACDC.

This evaluation used simulated authoritative decompositions [104]. The rationale for this approach was that using a small number of case studies exposes the study to potential bias introduced by the case studies. Rather than using a case study and its authoritative decomposition, the LimSim approach used in Shtern and Tzerpos' evaluation uses the case study as a seed from which many random systems are generated. This is achieved by randomly modifying the MDG and its corresponding authoritative decomposition [104].

Evaluations of the authoritativeness of all clusterings show repeat findings of the same result; the algorithms perform well in some aspects, but poorly in others. Glorie et al.'s findings [40] demonstrate this phenomenon in an industrial context. As none of these clustering algorithms is clearly generally better than any other (when comparing similarity to authoritative decompositions), the choice of an algorithm to apply to the remodularisation problem is dictated by the flexibility and applicability (data required by the algorithm and resultant data it produces). The following section takes this into consideration and summarises the surveyed techniques and shows how, of the algorithms surveyed, Bunch's flexibility makes it the best starting point.

§ 2.6 Summary of Applicable Techniques

This thesis focuses first on the use of clustering techniques to remodularise state machines for the purposes of abstraction. The various applicable approaches discussed in this literature review are summarised in this section from the perspective of their suitability. This summary focuses entirely on direct possible applications for brevity, and groups the techniques by the aims and objectives as defined in Section 1.3.

2.6.1 Aim 1: Applying Remodularisation to State Machines

This survey has shown that the remodularisation problem is highly related to the state machine hierarchy generation problem. The difference in the approaches indicates that there are a number of techniques that may be transferred and applied to the state machine clustering problem.

The remodularisation algorithms surveyed can be divided into several distinct groups:

Clustering Items are grouped by a similarity measure

Pattern-based Items are placed using pre-defined architectural patterns

Concept-based Items are grouped by concepts using formal concept analysis

Of these approaches, clustering and pattern-based approaches produce hierarchies, while concept analysis can be used to produce different partitionings that group elements by common concepts. Thus, concept analysis algorithms are not considered as a hierarchy is desired. Additionally, the use of patterns requires a set of known patterns to apply. This technique is highly similar to the transformation approaches already applied in state machine complexity reduction which has problems with computational complexity. Clustering, the only remaining technique is then chosen as the focus of the thesis.

Of the remodularisation techniques surveyed, a large majority operate on some form of graph derived from the system. In some cases this graph is extended to include multiple types of information, such as the graph of functions and types used in Arch [100] or the Resource Flow Graphs used in Rigi [83]. As these features do not exist in state machines, these techniques cannot be applied, however, other approaches that use only the graph may be applicable to the problem.

In order to apply a clustering technique to the problem, the following elements form part of a solution and are avenues of investigation:

Algorithm The process by which elements will be grouped together, including the type of linkage in hierarchical algorithms

Criterion The measure that will be used to assess candidate results, such as merges in hierarchical algorithms and how it will be mapped to numeric values, e.g. the similarity measure

The intersection of the available information for these models is the graph structure itself; edge and node labels are not present in MDGs and state machines respectively. This scenario, where only the graph is available (and its adjacency matrix) most closely resembles the modularisation of variables as described by van Deursen and Kuipers' [118]. Problems with the sparsity of the feature set and interpretation of dendrograms as identified by van Deursen and Kuipers [118] are therefore likely to persist if hierarchical clustering is applied to this problem.

In consideration of these problems with hierarchical clustering, other clustering approaches appear more suitable. The Bunch [81] clustering tool uses a hill climbing algorithm to locate solutions that maximise the MQ fitness function. As MQ favours solutions with fewer edges that enter or leave clusters, it has the potential to produce clusters for state machines that are of high quality. Additionally, MQ is based on the concepts of cohesion and coupling, values used by Anquetil and Lethbridge to assess the results produced using hierarchical clustering algorithms [9]. Using Bunch allows these criteria to be maximised directly, rather than indirectly through a similarity metric.

Furthermore, using a search-based approach allows the clustering criterion to be changed by changing the fitness function, which allows the exploration of search-based solutions to the rule-matching problem to be conducted using the same environment. This experimentation is performed in Chapter 4.

Chapter 3 further investigates the use of Bunch to the state machine clustering problem and identifies issues with the approach it takes to hierarchy generation. Approaches to clustering using linear representations, including the label approach of Chis [21] and the "indirect" approach applied to the refactoring problem [101, 48] are explored in Chapter 4. These approaches are applied using fitness functions derived from remodularisation and state machine hierarchy generation approaches in order to explore various fitness functions, including a search based approach to the rule-matching superstate transition generating techniques discussed in Section 2.4.

2.6.2 Aim 2: Addressing Qualitative Issues with Automated Remodularisation

The second aim of this thesis focuses on the regular finding of inadequate results typical in remodularisation. The use of expert knowledge is widely reported in remodularisation [12, 90, 81, 83, 100] approaches, as well as in clustering approaches [120].

These approaches aim to tailor the algorithm to their users in order to produce better results. As found by Wu et al., automated remodularisation algorithms are not likely to reproduce authoritative decompositions [126], and an easy interface to tailor the algorithm's results to the application is essential. As Glorie et al. [40] found that while Bunch produced modularisations that were good in places, the lack of a means to refine the result made it unworkable as a solution to the remodularisation problem.

Chapters 5 and 6 investigate a means to enable modularisations to be refined, independent of the modularisation algorithm used in order to solve this problem of authoritativeness.

§ 2.7 Concluding Remarks

The same graph-based representations used by both module dependency graphs and state machines makes the application of remodularisation to the state machine hierarchy generation a viable option. Remodularisation techniques can use clustering algorithms to group elements by an estimate of similarity, a measure which, in some cases such as those using adjacency lists for the feature vector, applies directly to states in state machines.

Remodularisation has seen widespread effort to improve results in terms of the structural quality and accuracy of the result. The former can be increased through the use of better algorithms, such as selecting a better linkage criterion, however the latter relies on ensuring the algorithm replicates choices made by a human.

Thus, the former task of increasing metrics is an optimisation problem; for a given state machine, the goal is to select the arrangement of hierarchies such that the quality metric is maximised. For large state machines, the space of hierarchies is large, which motivates the use of a more efficient exploration of the search space with metaheuristic algorithms. Chapters 3 and 4 explore this application of metaheuristic search to the state machine clustering problem.

The second problem, of accuracy, has remained unsolved by automatic approaches. While increasing metric values may produce better results, a human providing a subjective judgement on the result is likely to make choices that a metric may be unable to predict. This problem is rooted both in the choice of the metric, but the exposure to subjectivity cannot be ruled out, thus this problem is likely to require human intuition. Chapters 5 and 6 explore this component of the hierarchy generation problem using a refinement approach, based on Wagstaff et al. [120]'s use of constraints to improve clustering algorithms.

Chapter 3

Remodularisation of State Machines for Abstraction

The previous chapter outlined existing approaches to state machine superstate generation and remodularisation. It detailed issues that exist with rule based approaches and discussed the similarities of MDGs and state machines, showing how remodularisation approaches that operate only on a graph may be applicable to the state machine superstate identification problem.

This chapter tests this hypothesis through the application of the Bunch remodularisation tool to produce a single level hierarchy for state machines. In contrast to the existing composite state generation approaches, Bunch is a metaheuristic approach to the clustering of a graph. Section 3.1 shows how it can be applied to the composite state generation problem.

The performance of this application of Bunch is evaluated on two case studies by comparing the output to expert decompositions. Results of these comparisons show Bunch is able to completely recover the structure of one case study and partially recover the structure of the other.

After establishing the feasibility of clustering state machines with Bunch, the approach it uses for hierarchy generation is investigated in Section 3.2, where evidence that the layered approach that it uses results in over-fitting is encountered. Section 3.3.1 surveys the hierarchies of several open source software systems and determines that unbalanced hierarchies do occur, and that Bunch's limitation to layered hierarchies precludes it from reproducing structures produced by the architects of some systems.

An approach to rectify this scenario using a post-processing step based on the mean fitness value of elements of the hierarchy is then evaluated, determining it can increase the mean fitness in approximately 30% of the produced solutions.

This chapter makes the following contributions:

1 An empirical study into the feasibility of applying the Bunch search-based remodularisation tool to state machines that shows Bunch is able to partially recover elements of expert decompositions;

2 Analysis of Bunch's layered approach to hierarchy generation, demonstrating that fitness values of earlier levels impact the fitness of later levels; and

3 Investigation of the use of random mutation to mitigate over-fitting in Bunch by producing unbalanced hierarchies.

§ 3.1 Modularising State Machines with Bunch

As discussed in Chapter 2, there is a large body of research directed at the remodularisation of software. All surveyed approaches for state machines, however, use rule-based matching over all subsets of the state machine [108, 23, 62] to identify groups of states to place in a composite state.

As shown in Chu et al.'s case study, the abstraction provided relies on the state machine exhibiting patterns that match them [23]. The SCHAEM technique [62], discussed in Section 2.4 also has problems with computational complexity; each pair of states is assessed for the superstate rule, meaning constructing hierarchies is costly, Systä et al. showed that finding the optimal assignment of groups is NP-complete [108].

The computational complexity and dependency on rule matching limits the application of these approaches. As reverse engineered state machines are likely to be large, a more efficient approach is required. The Bunch [81] clustering tool applies a meta-heuristic search to the highly similar MDG clustering problem. A search-based approach avoids problems with the large search space at a loss of the guarantee of producing the global optimum. It also gives Bunch more flexibility to examine some solutions that would not be considered by rule-based approaches.

Bunch operates solely on the connectivity of the MDG, which makes it flexible for application to other graph partitioning problems, such as assigning heaps to threads to reduce the cost of garbage collection in a multi-threaded application [24].

Modularisation Quality, the fitness function used in Bunch to estimate the quality of a candidate partitioning (discussed in Section 2.5.6), aims to reduce the number of edges between clusters. The quality of the result relies on the relationship between this edge-reducing heuristic and an expert's notion of what constitutes a good organisation of the state machine. The experiment described in this section aims to measure this relationship.

This section investigates this application of Bunch and aims to identify any potential issues and to determine how suitably Bunch clusters state machines. The study addresses the following research questions:

RQ 1 How closely is Bunch able to reproduce a known grouping for a state machine?

RQ 2 What is the relationship between MQ values and similarity to a known good solution?

3.1.1 Generating Bunch-compatible Input

In order to cluster a state machine with Bunch, it must first be transformed into the correct MDG format. Bunch uses a variant of the Pajek graph definition format [11], where only edge definitions are required. An example is given in Figure 3.1. Bunch includes support for weighted edges, although each edge is assigned a uniform weight in these experiments.

```
state1 state2
state2 state3
state3 state1
state3 exit
```

Figure 3.1: A simple state machine encoded in the Bunch MDG input format

Although Bunch operates on a directed graph, it performs a step of removing reflexive edges before it begins a clustering operation, as they are meaningless in the context of a dependency graph. However, reflexive edges are common in state machines, where an event occurs that does not cause the machine to change state.

The removal of these edges means that Bunch may be operating on a graph that is not identical to the original state machine. This could negatively impact MQ values where reflexive edges of states are not counted towards the number of internal transitions. For the purposes of these experiments, the reflexive edges are ignored during clustering. A post-processing transformation restores the lost edges, as well as edge labels, which aren't encoded in the Bunch input format. This step is purely cosmetic however, and does not have impact on the clustering, but is necessary for presentation of the result.

Figure 3.2 summarises the process of clustering a state machine graphically, including the restoration of data removed by Bunch.

Once the state machines are transformed into MDG files they can be loaded into Bunch and clustered as if they were MDGs. Other Bunch features are therefore available, such

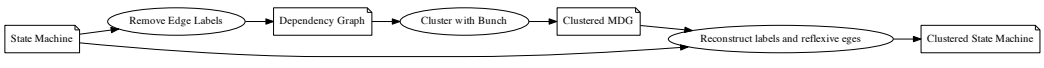
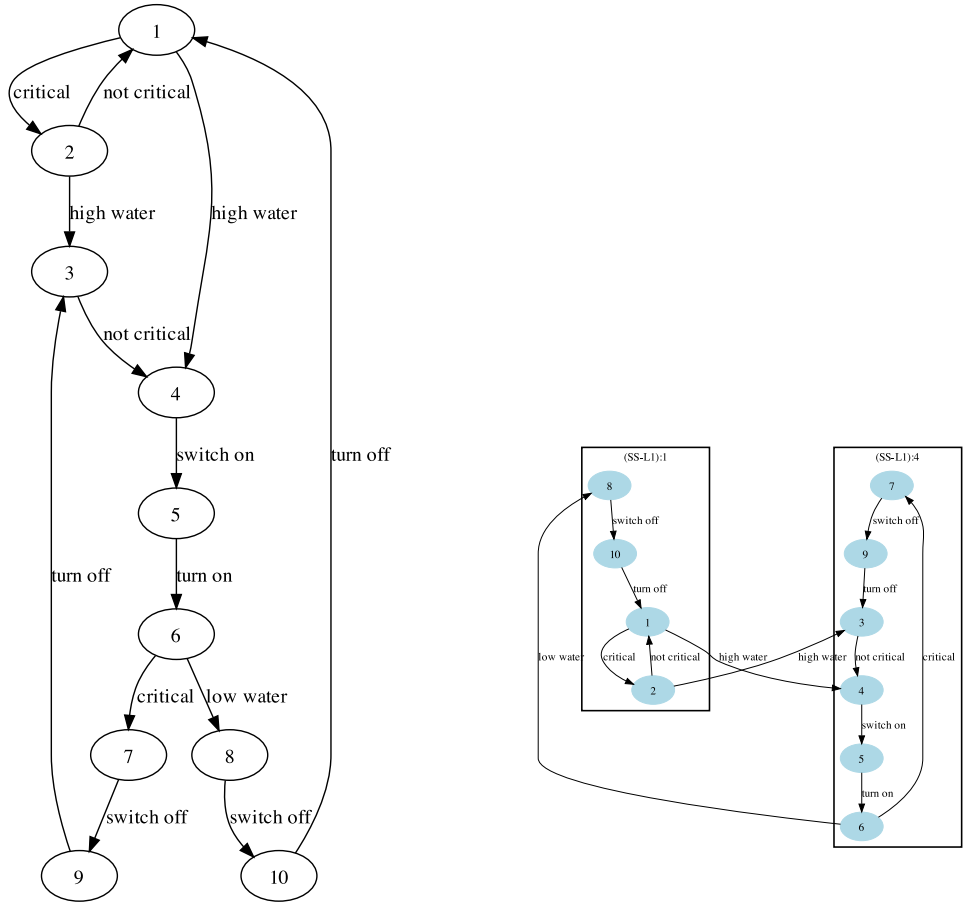


Figure 3.2: Modularising a state machine with Bunch



(a) The original state machine.

(b) The machine clustered by Bunch, with a division of the machine into superstates corresponding to high and low water levels.

Figure 3.3: The original and clustered state machine for a water pump controller

as omnipresent module detection and library detection. The features are not used in these experiments as they require domain knowledge to be used correctly and would constitute a potential threat to validity through the introduction of bias.

Figure 3.3 depicts a state machine for a water pump controller [28] (part **(a)**) and its corresponding clustering produced by Bunch (part **(b)**). The two superstates correspond to the water level; they group states for low and high water together. This structure was discovered by Bunch based solely on the connectivity of the graph, indicating that the structure alone is capable of leading the search to good decompositions.

3.1.2 Evaluation

The evaluation is conducted by comparing the clustering produced by Bunch to a known good clustering produced by a domain expert. This is an assessment of the "authoritative" performance of Bunch, as described in Section 2.2 which uses a similarity measure to evaluate the performance. Statecharts are used for the case studies, as they include a developer-generated hierarchy which can be used for the assessment. The similarity measures used are listed in Section 3.1.4.

State Machine Extraction

A state machine was first generated from the statechart of each case study. For statecharts containing multiple concurrent superstates, the largest superstate was selected. The statechart was then flattened by removing all superstates. Superstate transitions were transformed into individual state transitions from every state in the superstate. For example, if there is a superstate S containing a, b, c a transition $S \rightarrow z$ will be encoded as $a \rightarrow z, b \rightarrow z, c \rightarrow z$. Transitions to a superstate were rewritten to point to the default state for that superstate.

States were assigned unique names by prefixing them with all the names of the superstates that contain them to avoid name collisions if two states shared the same name.

Clustering

The extracted state machine's transition graph was loaded into Bunch and clustered as if it were an MDG following the process depicted in Figure 3.2. As Bunch is a randomised algorithm, the clustering process is repeated 30 times for each case study. This ensures the evaluation quantifies the typical performance and reduces the influence of an unusually good or poor result on the results.

Bunch produces a hierarchy of modularisations, forming "levels". The number of levels is not selected a-priori, but determined by the number of recursive clustering Bunch can perform until it produces a solution containing only one cluster. Each of the levels produced by Bunch was evaluated individually in these experiments.

3.1.3 Case Studies

The experiment was conducted on two case studies, both small state machines extracted from statecharts. The size of the case studies is a consequence of the lack of availability of larger statecharts.

Figures 3.4 and 3.5 show the `alarmclock` and `tamagotchi` case studies used in the experiment respectively.

The `alarmclock` example consists of 3 superstates; one for normal operations (display of time, alarm time), one for setting the current time and one for setting the alarm

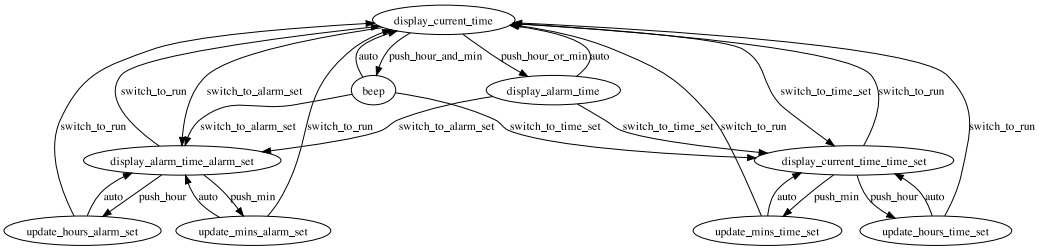


Figure 3.4: The `alarmclock` case study

time. It is taken from a conceptual model of a statechart used in a study on the use of finite automata to represent mental models [98]. The original statechart contained three concurrent superstates, this case study is the largest of the three.

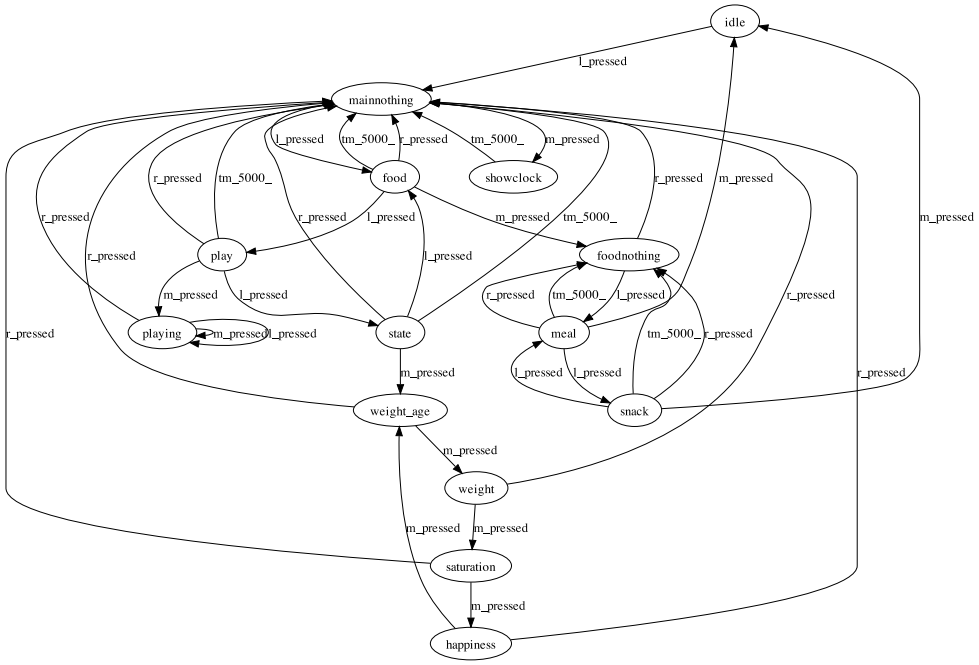


Figure 3.5: The `tamagotchi` case study

The `tamagotchi` case study is taken from a student project to produce a UML model of a 'tamagotchi' electronic pet toy [13]. It is a larger example, consisting of 7 superstates, including a main menu, a food menu, a status viewer and a catch-all superstate for all actions associated with playing the game. Two of the states in the machine appear outside of a superstate, for the purpose of the evaluation, each are placed in their own superstate, as it is not possible to encode entities outside of a cluster in Bunch. The impossibility of encoding a variable-depth hierarchy is analysed in Section 3.3.

3.1.4 Metrics

The following metrics were calculated for each level of individual result. The similarity measures were calculated between the expert decomposition and the result and estimate how close to the correct answer the Bunch result was.

MQ The fitness value of the clustering for that level

Depth Defined by Mitchell as the number of iterations of the hill climbing algorithm taken to produce the result at that level [78]. This is one of the metrics reported by the Bunch tool.

Evaluations The number of MQ evaluations taken for Bunch to produce the result. Bunch reports the total MQ evaluations for the whole MDG (rather than each level) so the value is the same for each level [78].

Runtime The amount of time required to cluster the MDG (in seconds). This is the aggregate time taken for all levels, so like the evaluation count it is the same for all levels.

Precision and Recall As defined by Anquetil et al. [9] and summarised in Section 2.2, precision and recall estimate the performance, in terms of the number of incorrectly clustered elements (precision) and the number of correctly clustered elements overall.

EdgeSim A modularisation-specific measure [78] and is discussed in Section 2.2. EdgeSim counts the number of pairs of edges which are the same in each clustering.

MeCl MeCl [78] calculates the number of merges of fragments of one clustering required to produce the other. It is discussed in more detail in Section 2.2.

3.1.5 Results

The research questions for this evaluation are addressed using inferential statistics on the measures described in the previous subsection. The following subsections summarise the findings for each case study.

The `alarmclock` Case Study

Summary statistics for the `alarmclock` case study are given for level 0 in Table 3.1. Bunch produced two levels in all cases, however only the results at level 0 were meaningful. Each of the results at level 1 consisted of a single cluster containing all elements.

Of the results for level 0 (Table 3.1), 10 were "correct", i.e. Bunch successfully produced the same modularisation as that selected by the designer of the case study. Higher values of MQ correlate with higher accuracy (Pearson $r = 1$, Spearman $r_s = 0.859$), suggesting that MQ is a good predictor of accuracy for this case study. A higher fitness

Table 3.1: Summary statistics for level 0 of the `alarmclock` case study

	MQ	Depth	MQ Evaluations	Runtime	Number of Clusters	Precision	Recall	MeCl	EdgeSim
Mean	1.44	4.67	158.00	1.60	2.33	66.67	100.00	96.00	83.33
Min	1.39	1.00	101.00	0.00	2.00	50.00	100.00	94.00	75.00
Max	1.54	9.00	204.00	8.00	3.00	100.00	100.00	100.00	100.00
St. Dev	0.07	2.38	23.57	1.45	0.48	23.97	0.00	2.88	11.99

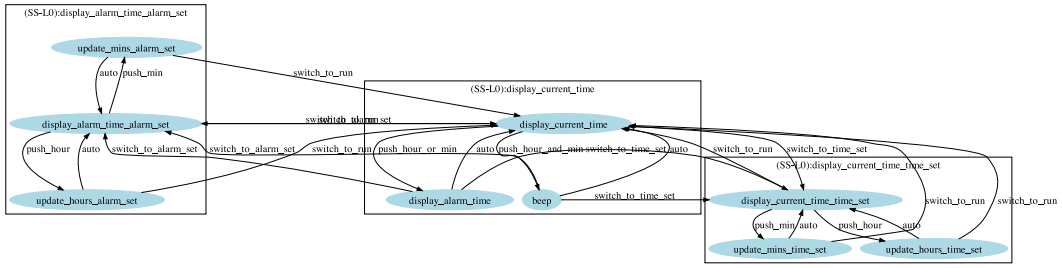


Figure 3.6: Correctly clustered `alarmclock` case study, with an MQ of 1.54

result from Bunch is shown in Figure 3.6, where the Bunch clustering was the same as the expert decomposition.

At level 0, two distinct classes of solutions were observed. In 10 of the 30 cases, the expert decomposition was exactly reproduced. In the remaining 20 cases, the solutions were poorer, with a precision of 100, recall of 50, EdgeSim of 94 and MeCl of 75.

The `tamagotchi` Case Study

Table 3.2: Summary statistics for level 0 of the `tamagotchi` case study

	MQ	Depth	MQ Evaluations	Runtime	Number of Clusters	Precision	Recall	MeCl	EdgeSim
Mean	2.27	16.27	559.03	2.57	5.53	73.94	46.22	91.33	61.33
Min	2.26	10.00	354.00	1.00	5.00	72.73	40.00	89.00	56.66
Max	2.28	23.00	760.00	9.00	6.00	75.00	53.33	94.00	66.66
St. Dev	0.01	3.40	107.97	1.76	0.51	1.15	6.77	2.54	5.07

Results for the `tamagotchi` case study are shown in Tables 3.2 and 3.3. As with the `alarmclock` case study, the final level generated by Bunch contains only one cluster and is not subject to additional analysis. In all cases, Bunch located 3 levels in the hierarchy.

The similarity to the authoritative decomposition for `tamagotchi` was lower than for `alarmclock`; Bunch never entirely reproduced it. The correlation between MQ and EdgeSim and MeCl values was again high at the highest level of detail (level 0): $r = 1$, $r_s = 0.865$ and was lower at the median level (level 1), $r = 0.540$, $r_s = 0.697$ for EdgeSim and $r = -0.540$, $r_s = -0.697$ for MeCl. This high correlation was also observed for the `alarmclock` case study, suggesting that the detail level is most suitable for authoritative results. This also supports the hypothesis that MQ can guide the search to authoritative decompositions (RQ2).

Table 3.3: Summary statistics for level 1 of the `tamagotchi` case study

	MQ	Depth	MQ Evaluations	Runtime	Number of Clusters	Precision	Recall	MeCl	EdgeSim
Mean	1.33	3.47	559.03	2.57	2.17	40.03	96.67	91.33	68.89
Min	1.27	1.00	354.00	1.00	2.00	34.88	80.00	91.00	63.33
Max	1.39	10.00	760.00	9.00	3.00	60.00	100.00	93.00	70.00
St. Dev	0.05	2.32	107.97	1.76	0.38	9.16	7.58	0.76	2.53

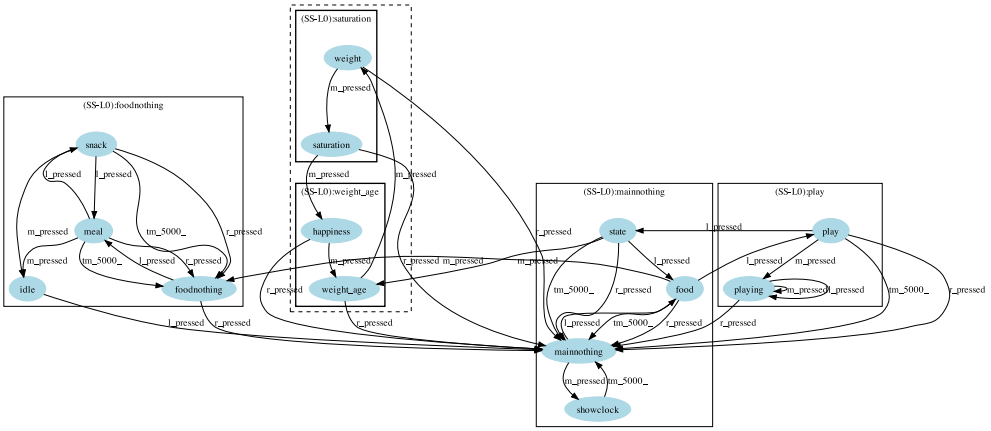


Figure 3.7: One solution produced for the `tamagotchi` case study

Figure 3.7 depicts the closest solution produced by Bunch. The majority of the states are clustered correctly, however the `happiness` and `weight_age` states were placed in a separate superstate to `weight` and `saturation` while the original decomposition placed them together (this is depicted by the dashed line in the figure). The two states that did not have any superstate (`idle`, `showclock`) were also placed differently. The `playing` state was also clustered as part of a superstate containing the `play` state, which again differs from the original.

The layered approach to hierarchy generation resulted in a detriment to Bunch's performance for this case study. A superstate in the expert decomposition was further divided at the lower level but appeared correctly at the median level (indicated by the dashed line in Figure 3.7). However, at the median level the other elements of the clustering were worse overall (depicted in Table 3.3).

Discussion

The results from the experiments suggest that Bunch is able to recover elements of an expert remodularisation. Although it never converged on the "correct" result for the `tamagotchi`, it was still capable of recovering some elements of it.

As Bunch operates solely on the connectivity of the graph, the good results for `alarmclock` are possibly due to its uniformity; there are a higher number of edges within superstates, an arrangement which MQ favours. The less uniform and larger `tamagotchi` case

study's decomposition contains features that MQ favours less than other arrangements, which possibly resulted in the poorer authoritativeness scores for this case study. The removal of reflexive edges from the graph prior to clustering may have contributed to the incorrect clustering of some elements. In addition, MQ awards no value to clusters that contain no internal edges. This makes it unlikely to produce singleton clusters, and in the `tamagotchi` case study, a singleton cluster in the expert decomposition containing the `playing` state was placed in another superstate with other elements (`playing` in the example in Figure 3.7).

The correlation at the detail level between MQ and the EdgeSim and MeCl measures does suggest that MQ models the design choices made by the developers who produced the expert decompositions. Although the correlation was lower at the high level, this result still suggests that Bunch and MQ are capable to an extent of approximating the decisions made by the developers. However, this study has also identified issues with its application to the problem. In addition to the singleton cluster problem, MQ is designed to minimise edges between clusters, which is contrary to the application of superstate transition rewriting, wherein multiple edges with the same label and target leaving a cluster are merged [108].

Further to the issues with MQ, the hierarchy generation process used by Bunch caused local improvement in authoritativeness to be conflated with an overall decrease in authoritativeness for the `menustate` case study. This, coupled with the problems with MQ suggests that, although Bunch can optimise elements of the decomposition, it does not present a complete solution to the problem.

In addition to the issues with Bunch itself, this evaluation demonstrated the difficulty in assessing the performance of Bunch, exacerbated by the limited availability of case studies with a decomposition. The similarity values for the top level of each case study were poor overall, however in both cases, the recall metric was 100% for all results. This is because recall looks only at the number of correct classifications, ignoring the number of clusters.

Threats to Validity

There are several potential issues which may be considered threats to validity:

There may be more than one authoritative decomposition. The conclusions are drawn from the similarity values with the expert decompositions. Although produced by experts, it is possible that another decomposition would be equally valid in the eyes of another expert. This problem is, however not unique to this experiment and is a threat to validity in all modularisation research that uses authoritative decompositions to evaluate performance.

The case studies are not likely to be representative of all state machines. The case studies may impart bias, firstly through their authoritative decomposition as previously discussed, but the small number and small sizes of the case studies do not permit these results to be extrapolated to all state machines. These results do not necessarily indicate typical Bunch performance, although they do offer

an insight into the relationship between MQ and the characteristics of a good organisation as viewed by the developer.

3.1.6 Conclusions of the Empirical Study

This study firstly shows the applicability of the Bunch algorithm to modularisation of state machines to produce superstates. The results do show that Bunch can recover elements of the hierarchy, echoing findings for the use of Bunch for automated software modularisation.

For this experiment, MQ appears to represent a reasonable approximation of what constitutes a good organisation, however the subjectivity of "good" makes the results of this experiment harder to interpret. It is therefore difficult to construct an objective judgement, a difficulty which is unavoidable for this problem, evidenced by Glorie et al.'s finding of poor performance of Bunch in an industrial setting [40].

Although feasibility was demonstrated by the experiment, issues were found with the hierarchy generation component of Bunch. For the `tamagotchi` case study, for example, one superstate in the hierarchy was further divided into two superstates. The next level, while correctly merging these two fragments also merged the other clusters which reduced the overall similarity of the result. Further difficulties were represented by the non-layered authoritative decomposition for the `tamagotchi` case study, which Bunch is unable to reproduce (this deficiency is further explored in Section 3.3.1).

This experiment has illustrated problems with Bunch's ability to recover the hierarchy for state machines and its hierarchy generation approach. The remainder of this chapter, and Chapter 4 focus on the latter problem, while the former problem of accuracy is investigated in Chapters 5 and 6.

§ 3.2 Over-fitting in Bunch

As identified in the previous experiment, the similarity of a Bunch result to an authoritative decomposition varies between levels in the hierarchy. The hill climbing approach used in Bunch is only able to modularise the system at one level at a time; hierarchy formation is attained by repeating the search on the modules identified by the previous search (this was described in Section 2.5.6). As Bunch is not guaranteed to produce the same result due to the stochastic nature of the hill climbing algorithm, repeated hill climbs may result in compounding poor performance.

The experiment in this section aims to assess to what extent this repeated search approach introduces error into the search, with the hypothesis that as the search begins at the lowest level, it may over-optimise the bottom of the hierarchy and restrict subsequent searches to solutions that are poorer than those reachable from less fit lower levels.

This section investigates the following research question: "Does the iterative hierarchy generation approach result in over-fitting?"

3.2.1 Method

The hypothesis is tested by measuring the fitness of individuals at the two levels in the hierarchy produced by the search. Fitness values are normalised according to the range of values observed over 100 searches for each case study. The process is as follows:

1 Cluster each case study 100 times

2 Remove all results that did not include at least three layers in the hierarchy

3 Normalise the fitness values for each level

Stage 1. Clustering Bunch is used to cluster each case study a total of 100 times. This yields 100 modularisations for the case study, each of which may have a different fitness values at different levels as Bunch is search-based. A total of 30 case studies are used in the experiment consisting of 13 MDGs extracted from various software systems and 17 finite state machines, of which three are synthetic.

Stage 2. Discarding of Flat Results Bunch may not produce more than one level of a hierarchy (ignoring the top level which always contains one cluster). These results are discarded from the set as there is no hierarchy to assess for over-fitting. The top level is ignored as it always contains only one cluster; the objective of this experiment is to determine if over-fitting occurs between two levels where there is more than one possible solution at each of these levels.

Stage 3. Normalising Fitness Values Fitness values at each level are normalised by scaling them into the range $[0 \dots 1]$ for each case study. This is achieved by collating the 100 fitness values at each level of each case study using the equation:

$$Norm(x) = \frac{x - Min}{Max - Min}$$

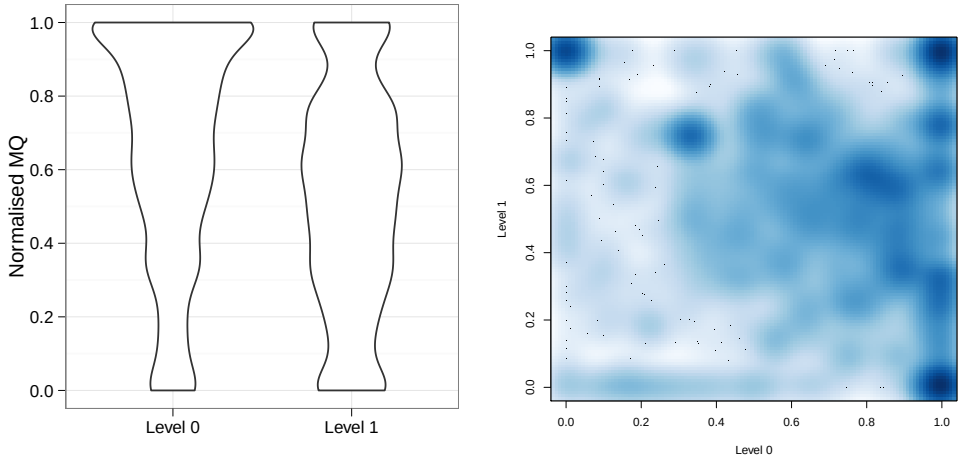
This scales the values to the range the observed values, such that the normalised value is 0 when the MQ value was the lowest of the observed values and 1 when it was highest.

3.2.2 Results

The process produced 2744 pairs of values of the 3000 clustering runs. This number is smaller due to the discarding of hierarchies with fewer than 3 levels described in Section 3.2.1.

Figure 3.8 depicts density and violin plots of the normalised MQ values (obtained as described in Section 3.2.1).

The violin plot (part **(a)**) shows the spread of the normalised MQ values at each level. The distributions are different for each level, with the proportion of greater MQ



(a) **(b)**

Figure 3.8: Violin plot and density plot of normalised MQ values, $N = 2744$

values being larger for level 0 than level 1. The level 0 performance alone shows that Bunch is able to identify a high proportion of high fitness solutions, however the poorer performance at level 1 suggests that the previous search predisposes it to produce less fit values. If no bias existed, the distributions would be more similar.

This apparent dependency is visualised in the density plot in Figure 3.8 part **(b)**. The darkness of each point corresponds to the frequency of those normalised MQ values at level 0 (x-axis) and level 1 (y-axis). An ideal result is a concentration of values at the point $(1.0, 1.0)$, however, there are concentrations around $(0, 1.0)$ and $(1.0, 0)$, as well as other points scattered across the range. These concentrations support the hypothesis of the research question, that layered hierarchy generation results in over-fitting.

The data shows a disparity between normalised fitness values at each layer and a dependency on previous fitness values. This supports the hypothesis that the repeated hill climbing approach used in Bunch may result in over-fitting. As Mitchell recommends the median level [78, p. 120] for analysing the solution produced by Bunch, several rounds of over-fitting may have occurred to produce the result, potentially lowering its fitness.

This section demonstrated the impact of the agglomerative repeated searching process on fitness of the hierarchies. The following section analyses the same method from the perspective of the types of hierarchies it is able to produce.

§ 3.3 Balanced Hierarchy Limitation

Results of the study in Section 3.1 for the `tamagotchi` case study show that the limitation of Bunch to producing only layered hierarchies precludes it from producing better solutions, as it is not possible for some clusters to be merged while others left as-is when producing the next level in the hierarchy.

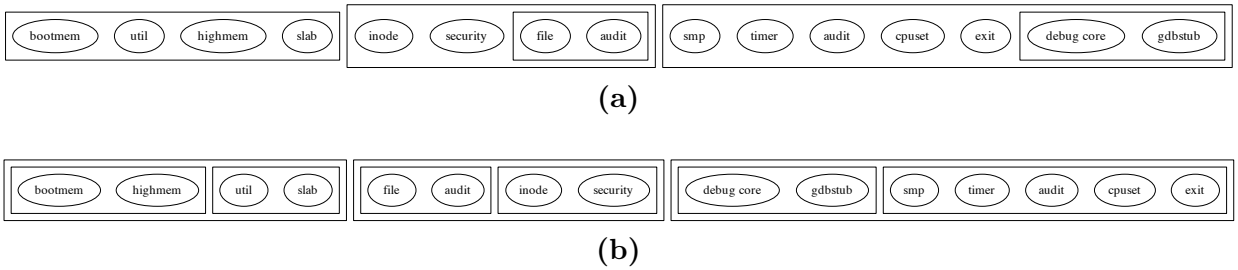


Figure 3.9: **(a)** Unbalanced, **(b)** balanced hierarchy

Figure 3.9 illustrates an unbalanced hierarchy (Part **(a)**) and the closest possible representation reachable by Bunch (Part **(b)**). The unbalanced tree in Part **(a)** shows two small modules at a lower level (`gdbstub` and `debug core`, `audit` and `file`) which Bunch would not be able to produce.

One possible approximation is shown in Figure 3.9 Part **(b)**, however this involves placing other elements deeper into the hierarchy than they would normally appear in a non-layered representation. Additionally, the module containing `bootmem`, `util`, `highmem` and `slab` is further divided, as MQ will award no value to singleton clusters.

This section further investigates this limitation in terms of the authoritative decompositions Bunch is unable to produce, determining that existing software systems contain structures Bunch cannot recover. It then concludes with an approach to improve mean MQ values of Bunch results by converting them to un-layered hierarchies through a random deletion operation.

3.3.1 Unbalanced Hierarchies in Open Source Software

As Bunch is constrained to only produce layered hierarchies, it may be unable to fully recover the hierarchy as intended by a developer if the developer did not use a layered structure. The result of the experiment in Section 3.1 shows that the similarity to the expert developer's solution could be improved if elements of the hierarchy could be merged while others remained the same. The hypothesis that hierarchies in existing software are not exclusively layered is tested by examining the hierarchies of five open source software systems.

Case Studies

Each of the case studies used in this survey are Java projects, which includes one tool and four libraries. The case studies were selected from the Ohloh [1] open source project aggregator's list of Java projects. Table 3.4 summarises the case studies used in this experiment.

[1] `http://ohloh.net`

Table 3.4: Case studies from the survey

Case Study	Version	Classes	Description
Collections	3.2.1	245	Apache Commons implementations of Java container classes
epubcheck	3.0-b4	75	Tool for validating `epub` ebook files
JDOM	1.0	57	XML parsing library
WiQuery	1.5-M1	204	Integrates the jQuery JavaScript library with the Apache Wicket web framework
ZXing	2.1-SNAPSHOT	184	Library implementing scanners for various barcode formats

3.3.2 Results

Table 3.5 shows the mean depth of each class for each of the case studies. In cases where this mean is an integer, each class appears at the same depth, corresponding to a layered hierarchy. If the mean depth is not an integer, then the classes appear at different levels in the hierarchy and can therefore not be represented as a layered hierarchy, and therefore cannot be recovered entirely by Bunch.

Table 3.5: Mean depths of classes in the each case study

Case Study	Mean
Collections	4.71
epubcheck	4.00
JDOM	2.63
WiQuery	5.16
ZXing	4.79

The values in Table 3.5 show that only one of the five case studies, epubcheck, has a layered package structure. This confirms the hypothesis that the hierarchy generation method used in Bunch would be unable to reproduce the package structures of all but one of the systems surveyed. Although this survey was conducted over a small number of software systems, it still indicates that there exist software systems where Bunch would never recreate the authoritative decomposition, even if it maximised MQ.

§ 3.4 Creating Unbalanced Hierarchies to Improve Bunch Results

The previous section demonstrated that Bunch is incapable of reproducing some hierarchies that are observed in existing software. As described in the previous section, some hierarchies can be mapped to a layered representation, but this inserts artificial

elements into the hierarchy. The result for the `tamagotchi` case study highlighted the detrimental effect on similarity to an expert decomposition that can result from this approach.

A means of removing these superfluous hierarchy elements is investigated in this section. Random elements of the hierarchy are deleted to produce an unlayered version of the hierarchy, which are assessed using the mean MQ value of all clusters in the tree. This section concludes with an evaluation on the five previously introduced case studies that demonstrates an improvement in mean MQ for at least 30% of the 100 mutants produced for each case study.

3.4.1 Random Node Deletions

Assuming the layered hierarchy produced by Bunch contains additional elements which place some parts of the MDG deeper than necessary, i.e. the hierarchy contains superfluous elements which add no value, it may be possible to produce an improvement by locating and removing these elements.

The random deletion approach works as follows: A hierarchy, in the form of layers produced from Bunch is first converted to produce a tree. An element from this tree is then selected at random (excluding the head node). Each child cluster from this element is then promoted to its parent. This produces a tree with one branch shorter than the rest. The resultant tree is then compared to the original tree using mean MQ (described later in Section 3.4.2), where an increase in mean MQ corresponds to an improvement.

3.4.2 Mean MQ

An estimate of the fitness of the unbalanced hierarchy is made by calculating the mean MQ of all clusters in the tree. This is based on the hypothesis that elements in the hierarchy with low MQ values detract from the overall quality of the hierarchy, increasing the number of clusters with little objective value. Thus, removing these clusters will increase the overall mean MQ and therefore the quality of the hierarchy.

Mean MQ is calculated by firstly summing the "internal" MQ values for all elements that contain at least one child cluster. Figure 3.10 shows an example hierarchy. In this example, clusters 0, 1 and 2 are used for the mean MQ calculation, as they are the only clusters that encapsulate other clusters.

The MQ values of each of these clusters is calculated using the normal MQ function. Each cluster is treated as a separate MDG; edges that leave or enter the cluster from clusters outside of it are ignored for the calculation. The sum of these internal MQ values is then divided by the number of parent clusters to produce the mean MQ value. In the example given in Figure 3.10, the mean MQ value for this hierarchy would be the sum of the MQ values of clusters 0, 1 and 2, divided by 4.

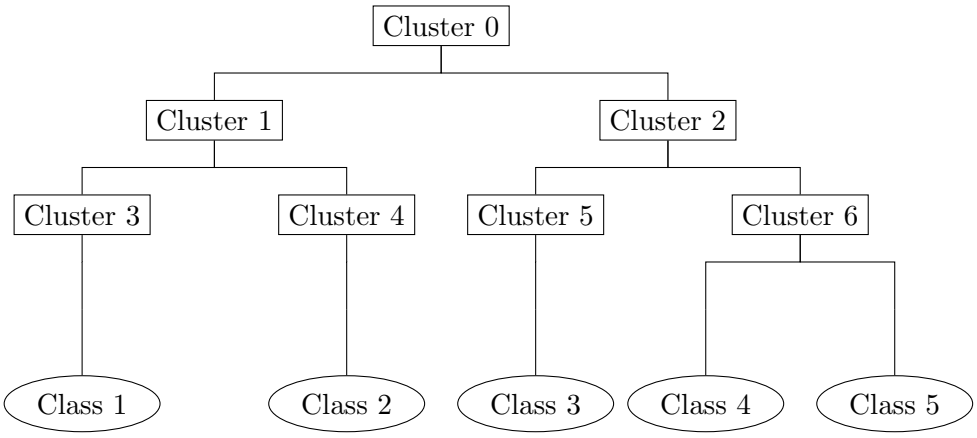


Figure 3.10: Layers in an unbalanced hierarchy

The mean is used as MQ favours higher numbers of clusters; each cluster's contribution to a layer's MQ is a number between 0 and 1. Higher numbers of clusters allow MQ to be increased, mean MQ removes this incentive in order to estimate the quality of the hierarchy.

3.4.3 Methodology

This study tests the effectiveness of the random deletion approach for maximising the mean MQ metric.

Each of the subject systems used in Section 3.3 was clustered in Bunch 30 times. The random deletion approach was applied 100 times to each case study to produce $15,0000$ mutants. The mean MQ values were calculated before and after the mutation was applied and used to determine if the approach was able to increase mean MQ values.

3.4.4 Results

The results from the random mutation experiment are summarised in Table 3.6. This table gives the percentage of mutants which had an increase in mean MQ values, showing that a minimum of 32% of the non-layered mutants had an increased mean MQ score compared to their layered original versions. A scatter plot in Figure 3.11 shows the original and mutant mean MQ values for each case study. The line, given by $y = x$ denotes improvements; instances where mean MQ was greater appear above the line.

This experiment has shown that a post-processing step may be used to produce an improved mean MQ score for a hierarchy produced by Bunch. The validity of mean MQ has not been tested, however. Thus, these results do not necessarily show that the Bunch results have been objectively improved upon. These results do, however, show how a transformation step may be used to modify the layered hierarchy produced by Bunch to reduce the impact of its inability to represent all possible hierarchies.

Table 3.6: Mutants with higher Mean MQ values

Case Study	Mean MQ Greater
collections	39.20%
epubcheck	32.20%
jdom	35.83%
wiquery	37.17%
zxing	37.97%

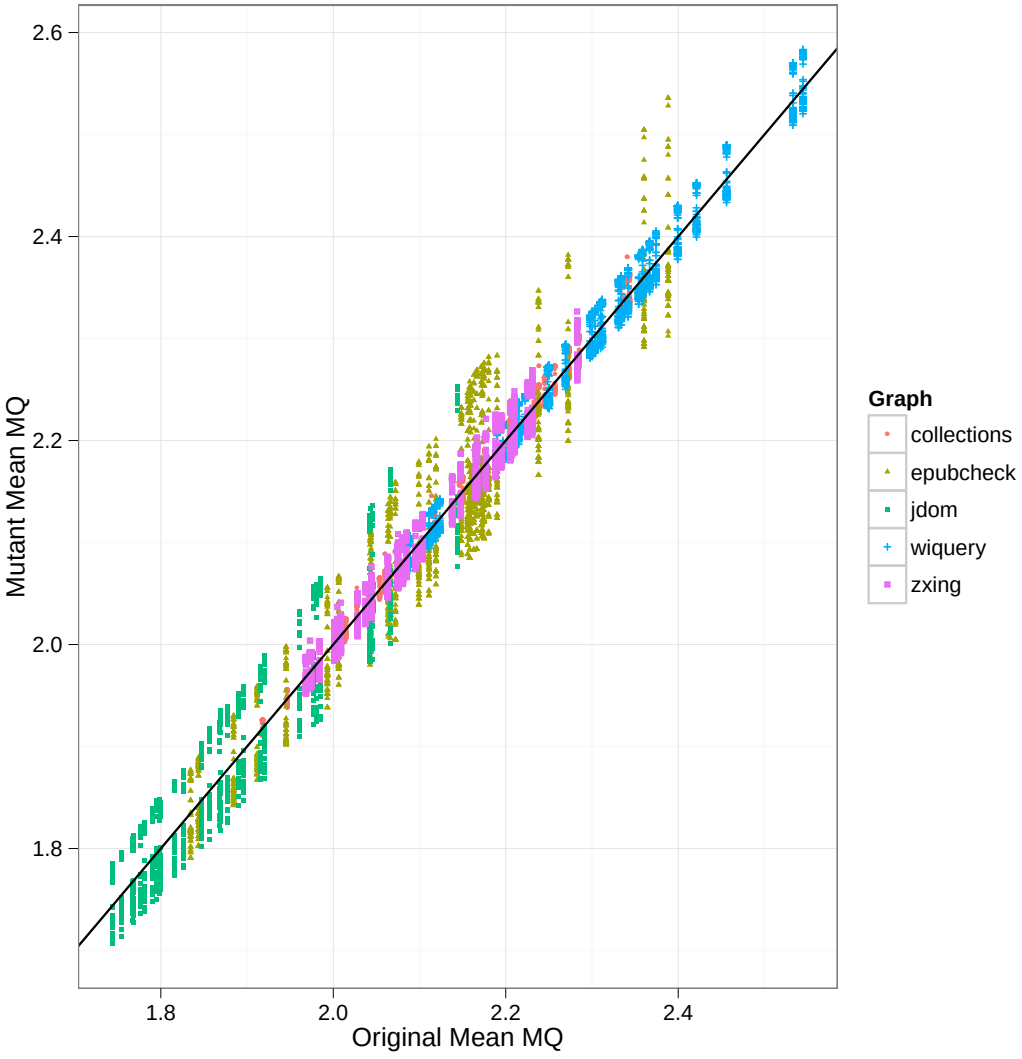


Figure 3.11: Scatter plot of mean MQ values for originals and mutants

§ 3.5 Conclusions

This chapter has demonstrated how Bunch can be applied to the state machine remodularisation problem. It has shown that the results obtained by Bunch are similar to the decompositions produced by the experts for two case studies, although in the larger example it did not completely recover the expert decomposition. The correlation of MQ with similarity to the authoritative decomposition suggests that it may be a useful measure to estimate the quality of decompositions for state machines. The issue of accuracy is revisited in Chapters 5 and 6, where human knowledge is applied to improve on the results yielded by Bunch.

Both state machines and MDGs exhibited the over-fitting problem caused by iterative level-by-level searches. As a result, Bunch results may be less fit at any level beyond the bottom level. For large systems, where many layers are produced, the recommended median level may be lower in fitness than it would be otherwise as a result.

In addition to causing over-fitting, the layered approach was found to constrain Bunch to producing only a subset of all possible hierarchies. A survey of five existing Java systems found that developers do not regularly produce layered package hierarchies. This has the effect that Bunch will not reproduce some solutions chosen by developers and demonstrates that an "ideal" remodularisation algorithm should be capable of producing a hierarchy with variable depths.

The application of a random deletion approach was demonstrated to produce unlayered mutants of Bunch produced hierarchies in approximately 30% of cases, using mean MQ as an estimate of the overall quality of the hierarchy as a whole. However, the uncertainty around this metric and the low success rate, combined with Bunch's over-fitting issue motivate an alternate line of enquiry that does not rely on repeated hill climbs.

This line of inquiry is continued in the following chapter, which investigates the use of a search-based approach that is able to search the entire hierarchy at once to attempt to avoid over-fitting and allow the generation of variable-depth hierarchies.

Chapter 4

Approaches to Search-based Hierarchical Clustering

The previous chapter developed the hypothesis that clustering can be used for the superstate identification problem. Results in Section 3.1 showed that Bunch is able to reproduce elements of developer-produced organisations of the case studies used in the experiment. However, the experiments in Sections 3.2 and 3.4 indicated that the layered approach to hierarchy generation causes over-fitting and prevents Bunch from reproducing some structures that are commonly found in existing software systems.

This chapter focuses on the latter issue, of hierarchy generation, in search-based clustering of both state machines and module dependency graphs. It describes and evaluates a novel application of genetic programming and transformation representations to clustering for state machines and MDGs. These approaches, in addition to a linear hierarchical representation are implemented in a tool, Crunch. These representations enable it to search for *hierarchies* represented by trees, as opposed to the agglomerative layering approach adopted in Bunch.

This approach avoids over-fitting caused by multiple searches and can produce a wider set of possible hierarchies including unlayered solutions. Crunch provides a flexible framework in which to implement fitness functions for hierarchies, as well as alternative representations of hierarchies in the search. The chapter begins with a discussion on methods to represent hierarchies in metaheuristic algorithms and describes approaches implemented in Crunch.

Following from this description of Crunch, it is evaluated in three experiments. The first experiment measures the performance of the representations implemented in Crunch and Bunch for partitioning 18 graphs. It determines that the representations that encode hierarchies produce lower MQ values than those that only represent partitioning and that Bunch's hill climbing approach produces higher fitness values than both the genetic algorithm and random mutation hill climbing search algorithms used by Crunch.

In the second experiment, the hierarchies produced by Bunch's normal agglomerative approach and MQ are compared to those produced using a hierarchical adaptation of

MQ with Crunch. This experiment finds that Crunch can produce higher MQ values at higher levels in the hierarchy, in contrast to Bunch which produces higher MQ values than Crunch at the bottom level of the hierarchies and higher overall total MQ values, suggesting that the two tools optimise the hierarchy in different ways.

The chapter concludes with the introduction of several fitness functions for clustering state machines, and compares the partitionings they produce when used with Crunch. It identifies several critical features required of a fitness function, and demonstrates that despite the apparent ability of MQ to recover developer-produced hierarchies, it does not produce solutions that enable superstate transitions to be rewritten. Further analysis of the structure of the hierarchies identifies difficulties with prioritisation of features. These identified difficulties make the case for incorporation of human knowledge into the clustering process, which is discussed in Chapter 5.

This chapter makes the following contributions:

1 Applications of label-based, genetic programming and a transformation-based representations to the hierarchical clustering problem, which is able to avoid over-fitting by searching for a hierarchy in one search.

2 Empirical evaluation of this technique in comparison to Bunch at one level and several levels in a hierarchy, finding that Crunch is able to produce hierarchies that have a higher MQ value at the top than Bunch, but a lower overall MQ value.

3 An empirical study of the performance of several fitness functions for the superstate identification problem, which finds that MQ and an information theoretic fitness function [69] for remodularisation produce assignments which reduce the number of edges between clusters, but does not identify solutions which can be rewritten with superstate transitions.

The chapter begins with an overview of evolutionary approaches that require or allow a hierarchy, in the form of a tree, to be represented in the search (or the genotype of an evolutionary algorithm), which are built upon in the description of the Crunch search-based hierarchical clustering framework in the subsequent chapter.

§ 4.1 Avoiding Over-fitting in Hierarchy Construction

Bunch encodes a clustering as an associative mapping of a cluster number to each file [78]. The file's position is used as an index into a vector which gives the cluster it belongs to. This list-based representation is not able to encode full hierarchies, so Bunch performs an agglomerative repeated clustering process to produce hierarchies level by level. Repeated layered searches can result in over-fitting, as demonstrated in the experiments in Section 3.2. Additionally, the method used to produce these hierarchies does not allow clusters to persist between levels—they must always be clustered at each subsequent level.

While existing hierarchical clustering algorithms repeatedly cluster the clusters identified at other levels, they are deterministic (assuming no ties of similarity measures) and perform merges of clusters one-by-one, producing a dendrogram. Bunch, in contrast, makes non-deterministic choices (due to the metaheuristic approach) and merges multiple clusters at each iteration.

4.1.1 Searching Hierarchies

The problem of remodularisation is an archetypical application of a search-based solution to a software engineering problem: the search space is large, candidate solutions are easily generated and evaluation of these candidate solutions is possible, using an estimation of quality [45], such as MQ as used by Bunch [81]. Bunch adopts such an approach, which makes it flexible and allows issues with the size of the input to be avoided, for example through the use of a user-supplied time limit [78, p. 242].

This flexibility makes the search-based approach an ideal application to the problem. Removing the repeated search for layers, the culprit of over-fitting in Bunch, in a search-based solution would allow hierarchical clustering to be performed without the limitations imposed by Bunch. One way of achieving this is to use the algorithm to optimise the whole hierarchy in one, rather than piece-wise over several searches. An approach achieving this goal requires two components:

- A representation expressive enough to represent whole hierarchies; and
- Fitness functions capable of evaluating *hierarchies*.

There are several approaches to the first component, including genetic programming and genetic clustering methods. These techniques are elaborated in the following sections.

Genetic Programming

The objective of encoding a hierarchy to the search can be achieved through the representation of the individual as a tree. Genetic programming techniques exist that evolve expressions in the form of abstract syntax trees [27, 60]. In genetic algorithms, these trees require specific operators to ensure conformance to the grammar of the expressions to be evolved. Koza [60], for example, defined crossover as a random exchange of subtrees between parents. This operator allows the conformance of offspring to the grammar to be guaranteed if the parents conform to the grammar.

Clustering places constraints on the tree that do not apply to abstract syntax trees used in GP [68]. Trees must only ever contain one instance of each node; trees that duplicate or remove nodes are not valid hierarchies in the clustering problem. Were random subtree crossover applied, the tree would need to be searched for these two invalid conditions and then repaired if they were identified. Figure 4.1 illustrates this problem; the two parents are crossed over at the highlighted points, producing the two

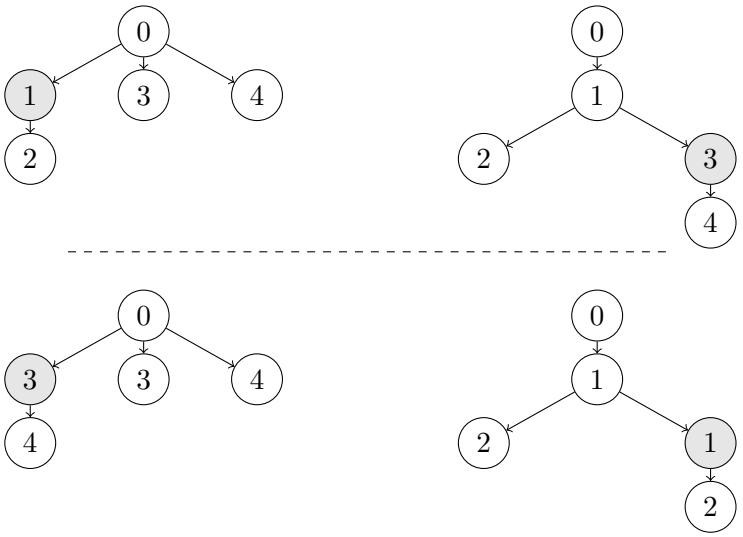


Figure 4.1: Random subtree crossover producing invalid trees

offspring below the dashed line, each of which has one duplicated element and and one dropped element.

Linear genetic programming avoids the use of tree representations in genetic programming [17, 91] by encoding the program as a list of operations, either in an imperative programming language [17], or as instructions interpreted on a stack-based virtual machine [91]. A linear approach allows genetic operators to operate over lists, rather than tree structures directly. These programs do not themselves represent trees as a list, but they represent abstract syntax trees which can be either parsed (for imperative programming languages) or converted to an infix tree using the "shunting yard" algorithm.

Although linear approaches enable hierarchies to be represented by the genotype whilst retaining standard crossover and mutation operators for linear sequences, they do not solve the problem of producing invalid trees directly. Additionally, linear genetic programs represent several statements in the individual, rather than a hierarchical syntax tree, although conditional operations have been implemented via jumping, for example. An approach based on genetic programming is described in the implementation of Crunch in Section 4.2.1.

Refactoring Representations

In search-based refactoring, the previous problem of preserving relationships in the tree persists. Harman and Tratt broadly categorise refactoring approaches as direct or indirect. Direct approaches use a combined genotype and phenotype, performing a similar process to GP where the structure is directly manipulated. In these direct representations that represent the tree in the genotype, care must be taken to ensure that the search does not produce solutions that encode invalid individuals as described in the previous subsection.

The alternative "indirect" method has been applied [101, 48] in which the genotype represents a series of transformations that modify the original system to produce the phenotype. Linear chromosomes can be used in this approach, which allow crossover and mutation to produce novel individuals without the need for any checking of the resultant output, provided the semantics of the transformations ensure that any sequence will produce a valid individual.

Hierarchical Evolutionary Clustering

Hierarchical genetic clustering has been achieved using an extension of the label-based representation described by Krovi [61]. This extension allows arbitrary levels of nesting to be encoded through the addition of a table of assignments of clusters to the chromosome [21]. This table, described in Section 2.5.2, includes a cell that may be referenced by other clusters in the table. The table must be extended to the maximum number of clusters that may be produced to form a full hierarchy for the input.

It is possible for this table to be placed in an invalid state. For example, a mutation may set a cluster's parent to itself or one of its children, forming a loop. In order to prevent this, Chis [21] applied a reparation step, wherein an invalid assignment (such as a loop) is broken by modifying all the children of the newly invalid mutant. Additionally, elements were only assigned to clusters that did not have any child clusters. If such an assignment arose, mutation was applied to remove it.

This approach requires that the mutation operator repair any invalid solutions it produces and also is reliant on the chromosome being extended to accommodate for potential assignments to clusters that may never need to be made. It is therefore possible that mutations applied to sections of the chromosome may produce individuals that offer no change in fitness.

This evolutionary approach, the transformation approach and genetic programming techniques are used in Crunch, a search-based hierarchical clustering tool that searches for full hierarchies in one run of a search algorithm, rather than using an agglomerative approach such as that used in Bunch. The next section details the architecture of Crunch and discusses the details of the implementation of the various types of representations it uses.

§ 4.2 Crunch

Crunch provides a flexible environment for the hierarchical search-based clustering of the two graph-based software models studied in this thesis: MDGs and state machines. Its design allows alternative fitness functions to be implemented and evaluated, using both a genetic algorithm and random mutation hill climbing search algorithm to evolve clusterings, using various linear representations.

Figure 4.2 shows the architecture of Crunch. The framework provides a means for implementing linear representations of hierarchies. These representations are decoded by

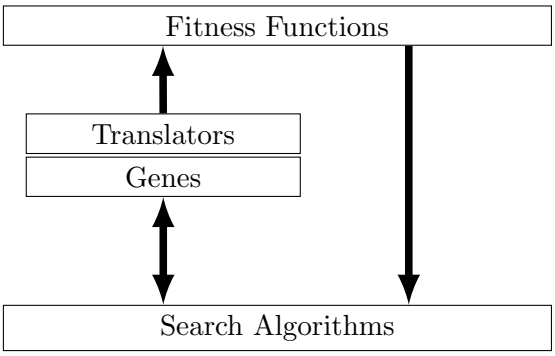


Figure 4.2: Crunch architecture overview; arrows represent data flow

a corresponding translator, before evaluation by the fitness function. Linear representation of the individual as a list of genes allows the edge cases associated with directly manipulating trees to be avoided—genetic operators such as crossover and mutation operate as normal on the list. Provided the phenotype is designed carefully (such that it does not duplicate or delete elements, for example), the consistency of the resultant trees in the phenotype can be guaranteed, allowing consistency checking to be skipped.

The gene representations implemented in Crunch can be divided into two categories:

GP Programs The individuals represent programs that are interpreted in a virtual machine which generates a tree for evaluation

Labels Individuals directly assign nodes to clusters, including a flat implementation defined by Krovi [61] and hierarchical version similar to Chis [21]

Crunch implements two forms of representation in a type of virtual machine. The first represents transformations as a sequence of operations in an imperative programming language. The other representation uses a stack to construct a tree. The following subsections describe these two implementations.

4.2.1 Hierarchy Transformation Representations

The first of the genetic programming approaches models each individual as a sequence of transformations that manipulate a tree. This approach is the same as that used in search-based refactoring [101, 48]. In clustering there is no initial solution, as with refactoring, so the phenotype starts as a clustering where each element is a terminal node. The transformations encoded in the genotype are applied to this tree to produce the phenotype evaluated by the fitness function.

Each individual in Crunch is a linear sequence of genes; in this case, each gene encodes a transformation, in the form of an instruction executed in a virtual machine. This approach carries over the benefits of simplified crossover and mutation from refactoring. The semantics of the instructions in the virtual machine are designed such that a valid

MOVE	CLUSTER4	CLUSTER3
NOP	CLUSTER1	CLUSTER6
NOP	CLUSTER7	CLUSTER4
MOVE	CLUSTER5	CLUSTER2

Figure 4.3: An excerpt of a linear GP clustering individual

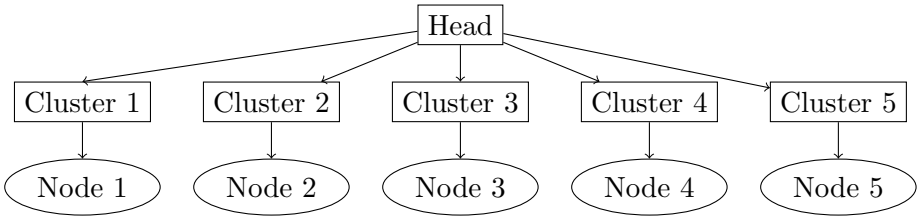


Figure 4.4: The starting state of the tree prior to modification by the clustering program

tree will always be produced, thus removing the need to repair or consistency check the phenotype.

Figure 4.3 shows an example fragment of a cluster program, a chromosome consisting of 4 genes. In this example, the symbol representation is used (described in Section 4.2.1). An example starting tree for a five element clustering problem is depicted in Figure 4.4. Each element in the input is placed into its own cluster, with these clusters all being made a child of the head node.

The Clustering Instruction Set

The language used in the virtual machine ensures all trees will be valid. Two operations are defined in the language, which both accept two arguments. Each argument is a number that represents a cluster in the tree, excluding the head node, or any of the basic elements (the number of clusters in the tree is fixed).

Two operators are defined for the language, `MOVE` and `NOP`. Each of the operators take two arguments, both integers, which identify clusters in the tree. The semantics of the operation are as follows, where `A` and `B` represent the two operands to the instructions:

`MOVE A B` Cluster B becomes a child of cluster A

`NOP A B` No operation

Unlike the approach adopted by Seng et al. [101], the programs that encode these transforms use a fixed length. This in turn requires that the representation be capable of encoding no transformation, as the starting configuration may be a valid solution.

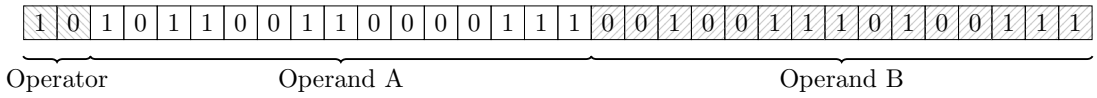


Figure 4.5: Assignments of bits to instruction components for the Integer and Bitfield genes

The `NOP` instruction allows this to be encoded. The parameters are retained in `NOP`s to ensure a mutation that reverses a `NOP` to a `MOVE` only modifies a single part of the program.

Crunch implements several different gene encodings that represent these instructions. Each of these encodings stores the same information, however the implementation of their mutation operators differ, which translate to different performance characteristics (demonstrated in Section 4.3). Each of the implementations is described in the following subsections. In all cases, the chromosome size (number of instructions in an individual) is set to the number of nodes, in order to allow a program that moves every node in the hierarchy to be represented.

Symbol Gene Representation The symbol representation encapsulates the three components (operator, operand A, operand B) into one gene object. Both of the operands are integers, limited to the number of nodes in the tree (given by the size of the input graph).

The mutation operation for this representation only modifies one of the three values, chosen at random. If the operand is selected for mutation, it is flipped: `NOP` becomes `MOVE` and vice versa. Operand values are given by the following equation:

$$NewValue = (OldValue + OldValue \times Amount) \mod TreeSize$$

The *Amount* variable is a parameter issued by the JGAP framework when a gene's mutation operator is invoked.

Integer Representation The Integer gene represents an instruction as a single 32-bit unsigned integer. The operator (`MOVE` or `NOP`) is placed in the 31^{st} and 32^{nd} bit of a 32 bit integer, while the two operands occupy the remaining bits, each taking 15 bits.

Figure 4.5 describes the assignment of values to the bits in the gene. The modulus of the tree size is taken of each of the operands when the gene is decoded by its translator to ensure the modifications always apply to valid nodes in the tree.

Mutation of this gene uses the JGAP implementation for IntegerGene [76]:

$$NewValue = OldValue + (MaxValue - MinValue) \times Amount$$

Where *amount* is a floating point number between -1 and 1, supplied by the mutation operator. The bounds on the integer are set at the maximum and minimum integer values supported by Java respectively. This mutation operation does not take the additional meaning of the specific bits into consideration, and thus multiple components in the gene may be modified through one operation.

Bitfield Representation In addition to the integer gene representation, Crunch also provides a bit vector implementation, which uses the built-in JGAP [76] bit vector gene with a size of 32. The assignment of values to bits is identical to the integer representation, described in Section 4.2.1. The mutation operator for this gene is a single flip of a randomly chosen bit. Unlike the integer representation, a single mutation will only change one of the three components of the instruction.

Limitations of the Virtual Machine

The linear transformation sequence representation, while allowing trees to be evolved is bounded in the number of clusters produced. This limitation is a consequence of the operand mapping used; if the number of clusters were variable, the nodes referenced by operands would differ between programs. The limitation to move-only operations also restricts the programs to a subset of all hierarchies, such as those where three elements should appear at the same level, the closest similar representation would still be able to place the three together in one cluster, albeit including several superfluous clusters beneath it.

A representation capable of producing arbitrary hierarchies requires the ability to add clusters to the tree. The following subsection describes how a stack-based approach can overcome this limitation. This approach contrasts to Chis' approach of ensuring the chromosome may represent all possible trees [21] through the use of a large, possibly empty table of pointers.

Stack-based Clustering

In order to remove the restriction on the number of clusters to allow arbitrary trees to be produced by Crunch, the problem of indexing clusters must be solved. The operands used in the previous approach cannot be extended in cases where the number of clusters does not remain constant.

The solution adopted in this representation is to encode the hierarchy as a sequence of merges of two clusters. This is achieved using a stack and a genetic programming approach.

The stack representation defines the following operations:

`PUSH A` Node A is placed on top of the stack.

`CLUSTER A` The top 2 elements of the stack are placed in a new cluster. The new cluster is placed on top of the stack. If there are less than 2 elements on the stack, the instruction is treated as `PUSH A`.

This implementation retains the execution of the program as a transformation. The `CLUSTER` operator causes a new node to be inserted at the root of the tree, and the top two elements of the stack to be placed beneath it. This ensures that a program that does not push all of the original elements on the stack still produces a valid tree.

1	1	3

Figure 4.6: A flat label chromosome

A `NOP` instruction is not implemented in the stack, as it is possible to achieve the same effect with a sequence of `PUSH` operations without corresponding `CLUSTER` operations. If multiple `PUSH` operations appear for the same operand, only the first is actually executed.

The stack representation is implemented as an encapsulation of both the operator (`PUSH` or `CLUSTER`) and the integer operand. As with the other program representations, the operand is limited to values between one and the number of nodes in the input graph. The maximum size of a stack program is where there is at least one `PUSH` for each element and there are sufficient `CLUSTER`s that the stack is empty at the end of evaluation. In Crunch, the chromosome size is set to double the number of nodes in the input to allow such a program to be encoded.

Summary of the Genetic Programming Approaches

The approaches described so far represent a transformation of a canonical starting tree into the resultant hierarchy through sequences of instructions encoded by the individual in the search. Various implementations of this approach were described for the imperative approach, wherein clusters may be made children of another existing cluster via the `MOVE` transformation. The limitations of the "virtual machine" defined included a limitation on the number of hierarchies; the stack representation was shown as an alternative that allows arbitrary numbers of clusters in the hierarchy and still permits standard crossover and mutation operators to be applied.

The following subsection details an implementation of Chis [21]'s approach to evolutionary clustering, based on a set of labels placed in a vector of integers.

4.2.2 Label-based Representations

Crunch also implements two representations that encode the tree in a more direct form, using an associative assignment of elements to clusters, as described by Krovi [61]. Each of the representations use this label-based representation, one is flat and the same representation used by Bunch [78] and the other extends this flat representation to encode a hierarchy, using the representation adopted by Chis [21]. In contrast to the GP based representations, these representations use the chromosome as a list of integers to represent the clustering; the parent of a node is given by the value at its position.

An example flat label representation of a small clustering of three elements is depicted in Figure 4.6. Classes 1 and 2 are assigned to Cluster 1 and Class 3 is assigned to Cluster 3.

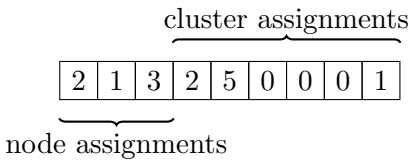


Figure 4.7: A hierarchical label gene

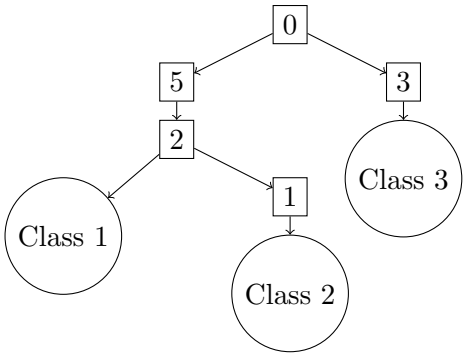


Figure 4.8: The tree for the chromosome in Figure 4.7

The hierarchical extension of the flat gene increases the chromosome size to include a region that describes parent assignments for clusters. This extension requires that the chromosome be lengthened such that there is sufficient room for the table to encode all possible hierarchies. There will be at most $2(n-1)$ elements in such a tree, so the total size of the chromosome is $n + 2(n-1)$, where n is the number of elements to be clustered.

Figure 4.7 depicts an example encoding of a tree, shown in Figure 4.8. Not depicted in the figure, cluster 6 is assigned as a child of cluster 1. This representation contrasts to Chis [21] in that there is no constraint on where a node may appear in the tree. Tree repair remains necessary, and is described in the following subsection.

Tree Repair

Trees constructed by the genetic algorithm are not guaranteed to be valid. For instance, a table may encode a cycle where a cluster is the parent of one of the nodes in its descendants. This is avoided by breaking these cyclic dependencies when they are encountered.

Orphans will be created when cycles in the tree are constructed, as illustrated in Figure 4.9. In this example, Cluster 1 is assigned as a child of Cluster 2, which is a child of Cluster 6. Cluster 6 is then a child of Cluster 1, resulting in the cyclic tree structure illustrated in Figure 4.10. This scenario can be avoided by removing any children that result in a cyclic graph. The removed children become orphans, which are added back to the root of the tree. This re-addition of orphans avoids the need for specific crossover and mutation operators, as used by Chis [21].

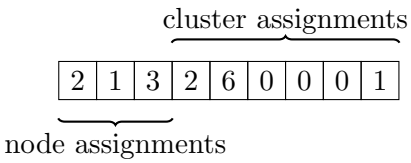


Figure 4.9: A hierarchical label gene which contains a cyclic dependency

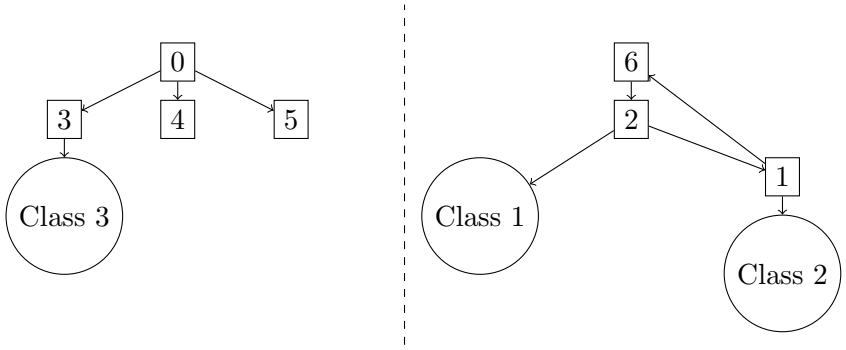


Figure 4.10: The tree for the chromosome in Figure 4.9, with an orphaned branch

4.2.3 Summary of Representations

Table 4.1 provides an overview of the gene representations described in the previous sections. In the Chromosome Size column, the variable n is the number of elements to be clustered, i.e. the number of states in the state machine or files in the MDG.

Table 4.1: Summary of the representations in Crunch

GENE TYPE	Gene represents	Hierarchical?	All Hierarchies?	Chromosome Size
Symbol		Yes	No (fixed number of clusters and Move only)	n
Integer	Instructions in a VM	Yes		n
Bitfield		Yes		n
Stack	Instructions in a stack machine	Yes	Yes	$2n$
Label	Parent of element in associative array	No	No	n
Hierarchical Label	Parent of element in associative array	Yes	Yes	$n + 2(n-1)$

4.2.4 Search Algorithms

Crunch implements both a genetic algorithm and random mutation hill climber. The chromosome representation and genetic algorithm components are built on the JGAP [76] Java genetic algorithm package. In both searches, the individual is a sequence of genes, with the length set according to the size of the input and the requirements of the particular gene, as summarised in Table 4.1.

Genetic Algorithm

Crunch uses the default genetic algorithm pipeline implemented JGAP [76] for the genetic algorithm search component. The default configuration is used, which includes a random single-point crossover operator, random point mutation operator and elitist ranked selector. A description of each operator is given in the following sections.

Crossover Operator

The crossover operator produces two offspring from two parents by swapping their sublists at a randomly selected point. Parents are selected at random.

Crossover is implemented using the default JGAP CrossoverOperator [1]. Parents are selected at random, and the operator produces offspring by exchanging their genes at randomly determined point.

The crossover rate is set to 35% by default. This rate is multiplied by the current population size to determine how many operations to perform at each iteration.

Mutation Operators

Each gene type defines a mutation operation; these conform to the `IGene` interface from JGAP [76] and are detailed in Section 4.2.1.

JGAP's mutation operator is applied to the population after crossover. It selects an individual gene from a chromosome at random and inserts a mutated copy of the chromosome containing it into the population.

The mutation rate is set to 12 by default, which represents a 1 in 12 probability that any gene in any of the chromosomes in the population is mutated. This results in $\frac{chromosomes \times population}{12}$ mutations occurring at each iteration.

Random Mutation Hill Climber

The RMHC implementation is given by Mitchell et al. [82] and previously described in Section 2.3.2. This algorithm uses the function given in Algorithm 3 to produce a mutated individual. The $applyMutation$ function referenced in this algorithm corresponds to the implementations for each of the gene types as described in Section 4.2.2.

4.2.5 Summary

This section introduced Crunch, a flexible architecture for clustering that uses various linear representations of trees to search the space of hierarchies. These linear representations are free of the limitation on layers imposed by the representation used by Bunch (with the exception of the flat label representation) and allow the space of all

[1] `http://jgap.sourceforge.net/javadoc/3.6/org/jgap/impl/CrossoverOperator.html`

Algorithm 3 RMHC mutation operator

```
function MUTATE(Individual)
    point ← random(0, Individual.size)
    amount ← random()
    sign ← 1
    if random(0, 1) >= 0.5 then sign ← −1
    end if
    Individual[point].applyMutation(random(), sign ∗ amount)
end function
```

reachable hierarchies to be searched in one run of the search. The following sections evaluate parts of the components of Crunch, starting with the performance of these different representations, the hierarchical performance of the Symbol gene, and finally an analysis of various fitness functions.

§ 4.3 Partitioning Performance of Crunch with MQ

Investigation begins with measurement of the performance, in terms of MQ values, produced by Crunch when it is configured to only optimise a single level of the hierarchy. Although some of the configurations in this evaluation represent hierarchies with more than one level, only the first level of the tree is used for fitness evaluation and performance assessment. In addition to comparison of the various search types and representations, results from Bunch are used as a further comparison method.

The research questions of this study are as follows:

RQ 1 What is the impact of using a hierarchical search on a single-level partitioning problem?

RQ 2 Which representation produces the best results?

RQ 3 Do any of the Crunch configurations exceed the performance of Bunch?

4.3.1 Methodology

In these experiments, 18 case studies are used, consisting of 9 state machines and 9 dependency graphs. Both state machines and MDGs are used in order to further test the hypothesis that remodularisation and state machine hierarchy generation are similar.

Fitness values produced by Crunch are compared to those produced by Bunch. Crunch was configured to stop the search after a limited number of fitness function evaluations were used; this limit was set at the mean number of MQ evaluations used by Bunch to cluster each case study over 30 repetitions.

Table 4.2: The case studies used in the experiments

Case Study	Description	Edges	Nodes	Evaluations
Module dependency graphs				
ttysnoop-0.12c*	TTY monitoring program	50	28	1240.57
compiler	Smaller compiler program developed at University of Toronto	32	33	2923.00
mtunis	Simple operating system written in the Turing language [47]	57	20	1305.27
ispell	Spell checking utility [47]	103	24	2548.77
rcs	Revision control system [47]	163	29	4926.17
bison	GNU version of the `yacc` parser generator [47]	179	37	9128.03
dot	Graphviz graph drawing tool	255	42	13166.70
grappa	Genome rearrangement analysis program [47]	295	86	67789.60
aalib-1.3*	Library for conversion of images to text-based ASCII art	313	109	59071.73
State machines				
liftController	Lift Controller [56]	24	5	77.83
congestion	TCP Congestion Control	25	5	63.40
bgp	The BGP routing protocol from RFC1771	64	6	109.87
atm3	Automated teller machine [56]	28	10	267.47
cvs	Version control system [65]	26	16	732.03
roomcontroller	State machine for a light control system [107]	34	18	1198.67
ssh	Secure shell protocol [93]	32	20	1133.07
collections†	`org.apache.commons.collections.HashBag`	38	26	2924.80
colt†	`cern.colt.map.OpenLongObjectHashMap`	63	53	22477.80

Reverse engineered MDGs and state machines are denoted by * and † respectively

The process of evaluating Crunch as follows for each case study:

1 Run Bunch on the case study 30 times and measure the number of MQ evaluations taken

2 Run Crunch on the case study 30 times, with the same MQ fitness function from Bunch, with an evaluation limit from Step 1

3 Compare the fitness values to assess performance

Case Studies

The 18 case studies used in the experiments are given in Table 4.2, sorted by size. The Evaluations column gives the mean number of fitness evaluations used by Bunch, which is used as the evaluation limit when running Crunch, discussed in Section 4.3.1.

Some dependency graphs were reverse engineered from software. In these cases they are marked by a $*$ symbol. These case studies were taken from the examples of output produced by the `cinclude2dot` script[2]. This script extracts dependencies wherever one C file includes another. The remainder of the module dependency graphs are taken from existing search-based clustering work.

State machines used in this evaluation are taken either from the StaMinA state machine repository[3] or automatically generated, denoted with a † in the 'State machines' section of the table. The automatic generation approach is discussed in Section 4.3.1.

[2] `http://www.chaosreigns.com/code/cinclude2dot/`

[3] `http://www.cs.le.ac.uk/people/nwalkinshaw/stamina/`

Generating State Machines

The state machines indicated as generated from "random usage" have been constructed using a sequence of random method calls on the class. The data used in these calls (where appropriate) is randomly generated.

Algorithm 4 Randomly call operations on `storage_class`

```
minSize ← 5
incrementSize ← 15
backingData ← new storage_class
dataSize ← minSize + random() * incrementSize
for i = 1 → dataSize do
    randomValue ← random()
    backingData.store(randomValue)
end for
underTest ← new storage_class
traceLength ← random() * backingData.count
for i = 1 → traceLength do
    call random_operation(underTest, backingData)
end for
```

Algorithm 5 *random_operation*(`underTest`, `backingData`)

```
dataItem ← randomElement(backingData)
randomMethod ← randomElement(methods_under_analysis)
call underTest.randomMethod(dataItem)
return
```

The listings in Algorithm 4 and 5 describe the process of constructing a collection of random data and using it as a source of arguments for repeated execution of random methods of an instance of the class. The `minSize` and `incrementSize` values are parameters to the algorithm, which were fixed at 5 and 15 respectively for each of the reverse engineered case studies.

Algorithm 4 was run a total of 100 times using AspectJ [59] to trace the methods executed as a result of each call. This produced a corpus of 100 traces of method call sequences, which was then transformed into a state machine using the k-tails grammar inference algorithm [16]. This performs merging of equivalent states to simplify the machine, as described in Section 2.1.1.

For the `collections` case study, the `org.apache.commons.collections.HashBag` class from the Apache Commons Collections library was subjected to the random execution approach. The HashBag class represents a collection which stores a value with a number which counts the number of occurrences of the value. The random data supplied to the class was a set of random keys and values placed in a hash table. These random values were used as parameters to the `add`, `remove` and `count` methods of the class.

The `colt` case study was reverse engineered from the Colt OpenLongObjectHashMap class (`cern.colt.map.OpenLongObjectHashMap`). This class represents a table which maps numbers to Object values. A hash table was used to store the random data, which consists of Java `long` keys and randomly generated `String` values. The `add`, `remove`, `get` and `keyOf` methods were called with this data.

Step 1: Clustering with Bunch

The first step of the experiment is to use Bunch to cluster each case study. This produces Bunch-derived decompositions of the systems and the number of evaluations of the MQ fitness function evaluations it took produce them. As it includes a random element, the Bunch clustering step is repeated 30 times for each study to ensure the values obtained do not impart bias in the comparison stage.

Step 2: Clustering with Crunch

Each of the case studies is then clustered using Crunch, using each of the gene types and both search types. For each gene type and search combination, Crunch is run using the a fitness evaluation limit given mean by the number of MQ evaluations taken by Bunch for the case study. As Crunch also involves randomised algorithms, each search is repeated 30 times to avoid the potential bias introduced if atypical results are produced by one search.

The fitness function used is the same MQ fitness function used by Bunch, made possible by an adapter that translates Crunch individuals to their equivalent Bunch representation.

Crunch is set to use the default parameters for both search algorithms as described in Section 4.2.4. The population size is fixed at 50 for the genetic algorithm search method (this was not tuned).

Step 3: Comparison of Performance

The clustering process in step 2 produces 30 clustered MDGs for each of the techniques used: Bunch and the 12 combinations of search types and representation used in Crunch. MQ values from each of these results are used for the purpose of comparison. In both cases the chromosome size is given by the function of the size of the input, which varies between the representations used; Table 4.1 summarises the sizes required by each representation.

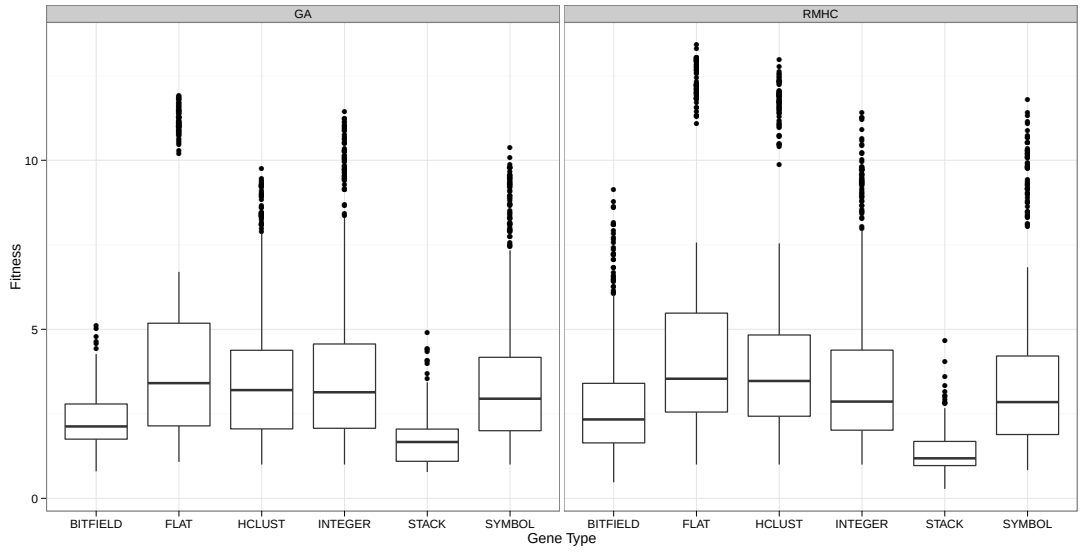


Figure 4.11: Box plot of the final MQ values for each search type

4.3.2 Results

RQ 1: Impact of using a hierarchical search on a single-level partitioning problem

A box plot of the final MQ values produced by Crunch for all case studies and all gene types is shown in Figure 4.11. The flat representation produced the greatest minimum and maximum values in both searches. Kruskal-Wallis tests of the fitness values grouped by gene type for each each search type produced chi-squared values of 783.133 and 1030.277 for the GA and RMHC searchers respectively. With 5 degrees of freedom, these translate to p-values below 0.001, indicating that there is a statistically significant difference between the MQ values produced by each representation.

Under the assumption that the distributions of each data set are the same type, two-sided Wilcoxon ranked sum tests were conducted between MQ values for the flat, non-hierarchical gene and the other hierarchical representations. The null hypothesis for this research question is that the MQ values from the hierarchical and flat genes are the same. In all but three cases, the median MQ value produced by the flat representation exceeded the median MQ values produced by the hierarchical representations. The three incidences of higher medians all occurred for the `ttysnoop` case study for the genetic algorithm searcher, with the hierarchical label representation (HCLUST), Integer and Symbol gene linear GP representations Wilcoxon p-values were < 0.05 in these cases. In the majority of cases, however it is possible to conclude that the hierarchical representation does result in lower median MQ values.

For this research question, the conclusion is that in the majority of cases, the hierarchical search will produce lower medians than a single-level partitioning search.

RQ 2: Relative Representation Performance

As shown in Figure 4.11, of the linear genetic programming implementations, the bitfield and integer genes produced the lowest MQ values overall. Table 4.3 provides a summary of the MQ values produced by each representation and search.

Table 4.3: Summary statistics for the flat experiments

Search Type	Gene Type	Min	Max	Mean	St. Dev	Median
Bunch HC	BUNCH	0.900	13.447	4.679	3.339	3.688
GA	BITFIELD	0.798	5.125	2.252	0.754	2.126
	FLAT	1.079	11.927	4.163	2.865	3.407
	HCLUST	1.000	9.750	3.622	2.046	3.202
	INTEGER	1.000	11.436	3.823	2.500	3.137
	STACK	0.778	4.907	1.688	0.663	1.666
	SYMBOL	1.000	10.369	3.543	2.175	2.947
RMHC	BITFIELD	0.474	9.119	2.798	1.643	2.333
	FLAT	1.000	13.420	4.581	3.161	3.538
	HCLUST	1.000	12.967	4.357	2.999	3.471
	INTEGER	1.000	11.417	3.668	2.394	2.861
	STACK	0.279	4.667	1.357	0.531	1.183
	SYMBOL	0.834	11.792	3.601	2.468	2.846

The hierarchical label gene (labelled HCLUST in the table) shows similar performance values to the flat label based gene, however the median, mean and maximum values are all lower. Both the Integer and Symbol linear GP genes produced similar fitness values, with the Integer representation producing greater mean and median values for both search algorithms. The Stack and Bitfield representations both produced the lowest fitness values of the set, seen in the box plot in Figure 4.11.

The performance of the random mutation hill climber, in terms of both median, maximum and mean values was higher than that of the GA for both of the label based genes. This result resembles the findings by Mitchell on the performance of genetic algorithms versus hill climbing for clustering [78], in which hill climbing was determined to produce superior fitness values, compared to a genetic algorithm.

The finding for this research question is that the performance of different genes has an impact on the fitness values attainable, with the Integer and Symbol GP genes being the best of the GP approaches in terms of median values.

RQ 3: Crunch Performance versus Bunch

Each of the median MQ values for Crunch search type and representation were compared with Bunch across each case study. At a significance level of $\alpha = 0.05$, statistically significant increases in median MQ were observed for the case studies and representation combinations shown in Table 4.4. Each of these cases where this result occurred are the smaller case studies of the data set and all are state machines.

Table 4.4: Statistically significant higher median MQ values

Search Type	Case Study	Gene Type	Crunch Median	Bunch Median	p-value
GA	bgp	BITFIELD	1.750	1.095	<0.001
	bgp	FLAT	1.750	1.095	<0.001
	bgp	HCLUST	1.708	1.095	<0.001
	bgp	INTEGER	1.750	1.095	<0.001
	bgp	STACK	1.708	1.095	<0.001
	bgp	SYMBOL	1.750	1.095	<0.001
	congestion	BITFIELD	1.750	1.000	<0.001
	congestion	FLAT	1.750	1.000	<0.001
	congestion	HCLUST	1.750	1.000	<0.001
	congestion	INTEGER	1.750	1.000	<0.001
	congestion	STACK	1.750	1.000	<0.001
	congestion	SYMBOL	1.750	1.000	<0.001
	liftController	BITFIELD	1.079	1.000	<0.001
	liftController	FLAT	1.079	1.000	<0.001
	liftController	HCLUST	1.079	1.000	<0.001
	liftController	INTEGER	1.079	1.000	<0.001
	liftController	SYMBOL	1.079	1.000	<0.001
RMHC	bgp	BITFIELD	1.645	1.095	<0.001
	bgp	FLAT	1.862	1.095	<0.001
	bgp	HCLUST	1.750	1.095	<0.001
	bgp	INTEGER	1.750	1.095	<0.001
	bgp	STACK	1.683	1.095	<0.001
	bgp	SYMBOL	1.683	1.095	<0.001
	congestion	BITFIELD	1.583	1.000	<0.001
	congestion	FLAT	1.750	1.000	<0.001
	congestion	HCLUST	1.750	1.000	<0.001
	congestion	INTEGER	1.750	1.000	<0.001
	congestion	STACK	1.583	1.000	<0.001
	congestion	SYMBOL	1.583	1.000	<0.001
	cvs	FLAT	4.417	3.881	<0.001
	cvs	HCLUST	4.167	3.881	0.018
	liftController	FLAT	1.127	1.000	<0.001
	liftController	HCLUST	1.103	1.000	<0.001

Figure 4.12 depicts a violin plot of the MQ values produced by each gene type for each gene type for both Crunch search types. It shows that the distribution of values with the bitfield gene more closely resembles the other genes, with a higher maximum value, for the random mutation hill climber compared to the genetic algorithm. This increase in performance is not observed in the case of the stack representation, suggesting that while the RMHC algorithm appears to produce better results overall, the representation is still the most significant factor.

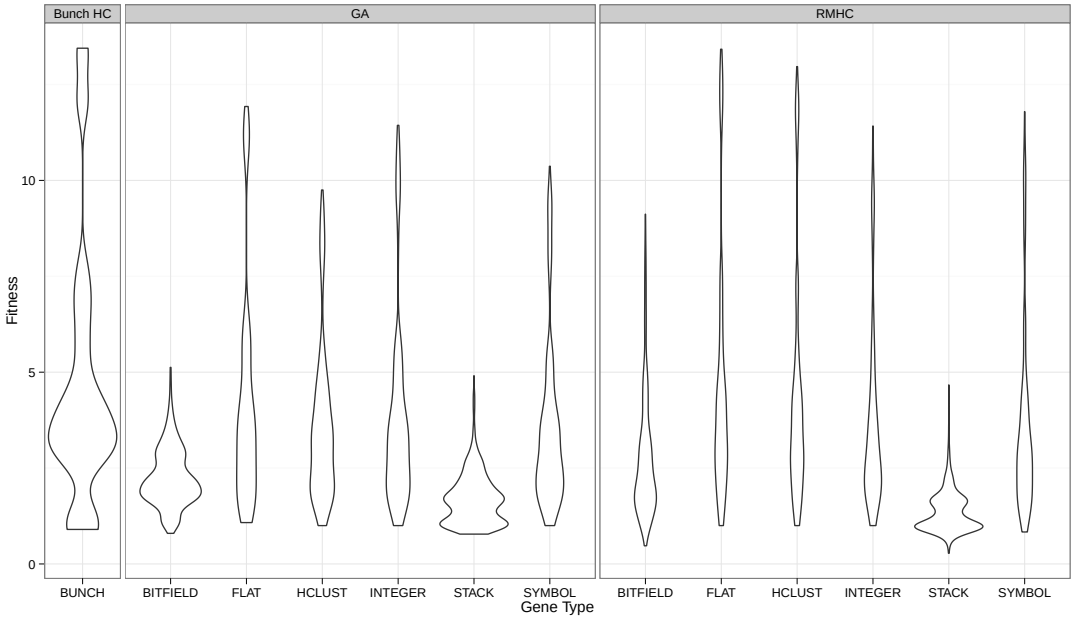


Figure 4.12: Violin plot of fitness values

The results for this research question have determined that Bunch is superior to Crunch for single-level partitioning for all small state machine case studies.

4.3.3 Discussion

The results show that the use of the hierarchical representation does impact the performance of Crunch when configured to perform a partitioning task. This is demonstrated in the data for RQ1, in which the flat label gene was only outperformed in 3 instances, and confirms that searching a larger space has a higher cost, in terms of the reachable fitness values.

Mutation operations on the bitfield gene only flip one bit at a time, resulting in very low overall changes in value. As the RMHC searcher, which searches entirely through mutation, yielded higher values, it can be concluded that mutation is essential for good results with Crunch, and that the "power" of individual mutations for the Bitfield gene is too low (only small changes can be introduced by it).

Results for the stack gene were also poor, with it having the lowest minimum and median MQ values of all the representations studied. The higher median for the genetic algorithm suggests that incremental improvement is facilitated by the genetic algorithm more so than the mutation imparted during the RHMC search. Dependencies on side effects may be the culprit of the poor performance of this representation as the order in which operations are carried out determines the hierarchy produced.

MQ values were similarly distributed in the flat and hierarchical label based genes. The reduction in maximum MQ values with the hierarchical gene could be attributed to the

increased search space; the chromosome must almost triple in size to accommodate the hierarchical label section. In turn, this expands the search space considerably. However, the equivalent stack implementation and this gene's close performance to the flat label gene suggests that this expansion of the search space is negligible.

Comparison of the performance against Bunch shows that in some rare cases Crunch was capable of producing higher MQ values even using the hierarchical representations. This occurred in all but the smallest case studies however, with the fewest number of iterations. In particular, the number of evaluations is sufficiently low that in the case of the genetic algorithm, very few iterations will have taken place. In their survey of the use of statistical analysis of results of randomised algorithms, Arcuri and Briand [10] note that an existing comparison using such a low evaluation limit may in effect be equivalent to a random search. As such, it is not possible to use this apparent positive result to conclude that Crunch exceeds the performance of Bunch in the general case. However, it does provide insights into the application of Bunch to small case studies and suggests Crunch may be able to locate superior assignments to Bunch due to its ability to explore hierarchies, even when optimising at one level.

The overall finding of this experiment, that Bunch is superior to Crunch at one level, is possibly due to the larger search space Crunch must traverse as it produces full hierarchies regardless of the level(s) the fitness function evaluates. This shows that Crunch is unsuitable in scenarios where only one level of hierarchy is required, especially for larger case studies. This limitation is by design however, it deliberately eschews single level performance in order to avoid the problems with hierarchy generation in Bunch identified in Chapter 3.

Threats to Validity

The use of randomised algorithms exposes the results of this experiment to several threats to validity:

Results may be unrepresentative The use of multiple repetitions of each of the search algorithms mitigates the chance that the results obtained are not typical of the algorithms surveyed in the experiments. Non-parametric Wilcoxon tests ensure that comparison of the medians is carried out only when doing so is statistically sound.

Low numbers of evaluations may not be exercising the genetic algorithm As the number of evaluations is fixed by Bunch's hill climber, and the population size is fixed at 50, it is possible for low evaluation budgets to be expended very early, possibly prior to one iteration of the GA. Thus, the results for these case studies do not characterise Crunch's performance in general, however the purpose of this experiment is to evaluate Crunch as an alternative to Bunch and thus the evaluation limit is necessary to allow that comparison to be made.

Higher MQ values may not equal better solutions The high MQ values identified, higher than Bunch in the cases described in Section 4.3.2, may not correspond

to actual improvements in the quality but be due to Crunch exploiting MQ to artificially increase it. However, this criticism is not unique to Crunch and extends to any optimisation of a metric.

4.3.4 Conclusions of the Partitioning Experiment

This experiment has demonstrated how the various components that Crunch is comprised of allow hierarchies to be searched to optimise the same single level MQ fitness function used in Bunch. It demonstrated that some of the genetic programming representations, specifically the Integer and Symbol genes perform similarly to the more direct hierarchical label based representation. Finally, the performance of both the bitfield representation and stack representations was shown to be considerably lower than the other representations used.

The next section extends upon this study to hierarchical results, using a similar experimental design.

§ 4.4 Hierarchical Performance of the Symbol Gene

This experiment follows a similar methodology to the previous experiment to investigate the performance of hierarchy generation in both Crunch and Bunch. In these experiments, only the Symbol genetic programming representation is used. MQ remains as an assessment criterion, however the assessment is conducted at multiple levels, rather than the one level used in the previous study.

The research question investigated by this experiment is:

RQ What is the difference in the hierarchies produced by the Symbol linear GP compared to Bunch?

4.4.1 Methodology

The design of this experiment is highly similar to the previous experiment, described in Section 4.3.1. The same set of case studies is clustered in Bunch, again using the number of MQ evaluations as the fitness evaluation limit in the Crunch searches. Evaluation is conducted in the same manner, using the MQ metric, although the assessment is conducted at multiple levels in the hierarchy.

The steps of the experiment are as follows:

1 Cluster the case studies with Bunch to produce hierarchies and MQ evaluation counts

2 Cluster the case studies with Crunch to produce hierarchies, using Bunch's MQ evaluation number as a fitness evaluation limit

3 Compare the hierarchies produced by the two tools

Stage 1: Clustering with Bunch

Each of the case studies is clustered in Bunch, in the same way as performed in the previous experiment. The case studies used in this experiment, detailed in Section 4.3.1, are those used in the previous experiment.

In this step, Bunch is used to produce a total of 30 hierarchies, in order to ensure the results are representative.

Stage 2: Clustering with Crunch

Crunch is then used to cluster each case study a total of 30 times to ensure the resultant hierarchies are representative. Both the random mutation hill climber and genetic algorithm searchers are used. In contrast to the previous experiment, the Bunch single level MQ fitness function is directly applied in Crunch. As Crunch generates hierarchies, and not partitions, it requires a fitness function to evaluate the whole hierarchy, rather than the single level evaluation provided by MQ. In order to compare the MQ maximisation performance of the two tools, Crunch was configured to use an adaptation of MQ. This hierarchical MQ (HMQ) function is described in the following subsection.

Fitness Function: A Hierarchical Adaptation of MQ

To estimate the overall quality of a hierarchy, HMQ sums the MQ values of each layer in the hierarchy. HMQ is then given by:

$$HMQ = penalty \times \sum_{i=1}^{k} MQ_i$$

Where k is the number of levels in the hierarchy and MQ_i is the MQ value of level i. The *penalty* term, further elaborated in the next subsection, is only set for Crunch results; for Bunch results it is set to 1.

Layered Hierarchies

While Crunch produces an unbalanced tree of clusters, Bunch produced a balanced, layered hierarchy. This is problematic as Bunch fitness functions operate on these layers, rather than individual clusters in the hierarchy. Calculation of MQ at layers rather than for individual clusters means that MQ evaluates a whole partitioning of the graph, meaning deeper elements in the clustering, where not all nodes are present cannot be evaluated without modification of MQ or the hierarchy. Figure 4.13 illustrates this problem: cluster 5 is the only cluster at its level in the hierarchy, and it does not partition the whole graph.

One solution to the problem is summing the cluster factors for all trees in the tree, although this would not result in a direct comparison of two MQ-optimising algorithms. The alternative, which is implemented in Crunch, is to treat the cluster tree as a

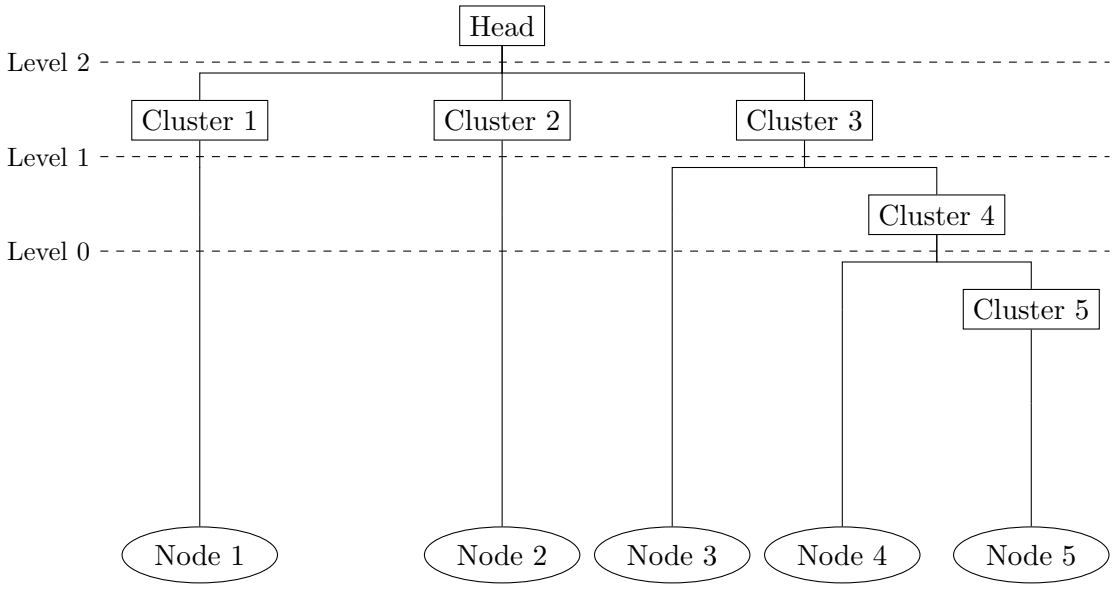


Figure 4.13: Layers in an unbalanced hierarchy

dendrogram, with each node extended to the bottom level. By doing so, each level in the hierarchy can be treated as a cut-point, which produces a partitioning of all nodes.

These cut-points are illustrated in Figure 4.13. Cuts are made from the first level that causes at least two nodes to be clustered together. In this example, Cluster 4 is the deepest in the hierarchy which has more than one node beneath it, so the cuts start from this point. In cases where clusters do not appear at lower levels, they are extended from the previous level. Although Clusters 1 and 2 only appear at level 1, their assignments are carried down to level 0, for example. Each layer is a group of clusters all with the same depth.

Transformation to a layered hierarchy allows layered fitness functions (such as MQ) to be used while still retaining some benefits of the use of a hierarchical search: it is possible for Crunch to represent individuals where only one cluster is divided at a lower level while others remain the same. Bunch is unable to produce such a hierarchy, a limitation found to negatively impact its performance in Section 3.1.

Although this treatment of the hierarchy as a dendrogram allows MQ to be used without modification in Crunch, the transformation to layered hierarchies potentially creates an exploitable weakness in the fitness function. As increasing the depth of one hierarchy results in more levels, it is possible to exploit this property to increase HMQ by maximising the number of clusters; as MQ is always likely to be non-zero (unless there are no intra-edges in any of the clusters at a level), the search may in effect multiply the MQ value of one layer by extending a long branch to produce more layers than necessary.

The *penalty* term in the HMQ definition in Section 4.4.1 is designed to limit the possible negative impact of this vulnerability by reducing the fitness values of individuals that

exceed a user-specified limit on the depth of the hierarchy. This penalty is given by:

$$penalty = \begin{cases} 1 & depth \leq penaltyAmount \\ 1/(depth-(penaltyAmount-2))^2 & \text{otherwise} \end{cases}$$

In this case, and for these experiments, *penaltyAmount* was set to 5.

Stage 3: Comparison Criteria

The quality of each hierarchy produced by both tools was assessed by calculating the MQ values at various levels, as well as determining the depth of it in order to characterise the result. As Bunch and Crunch produce hierarchies in different ways, the levels at which they are compared are defined as follows:

Top MQ The MQ value of the top-most layer of the hierarchy that contains the fewest number of clusters, excluding the head of the tree which contains one cluster. In cases where Bunch produces only one level (containing a single cluster), that level is used for the top level MQ calculation.

Bottom MQ The MQ value of the lowest layer in the hierarchy. In the case of Crunch results, this is the lowest level of the hierarchy where a cluster contains more than one element. Bunch may produce a single level of hierarchy in some cases; in these cases the bottom and top MQ values are the same.

Sum MQ The MQ value of each layer in the hierarchy is calculated and summed. This metric is the same as the HMQ metric described, in Section 4.4.1, without the penalty applied.

Depth The total number of levels in the hierarchy, including the root of the tree.

4.4.2 Results

Table 4.5: Depth values produced by each search type

Search Type	Mean	St Dev.	Min	Max	Median
BUNCH	3.143	1.043	1.000	6.000	3.000
GA	4.965	0.184	4.000	5.000	5.000
RMHC	5.006	0.262	4.000	7.000	5.000

Table 4.5 shows the depth of the hierarchies produced by each search type. The difference between the Bunch results characterises the difference between the two approaches; while Bunch has a higher variation in depth, the Crunch results are highly concentrated

around the depth limit imposed by the fitness function. In all cases, Crunch produced at least 3 levels of hierarchy, while Bunch produces as few as one, where the solution is one layer containing a single cluster. The values in this table suggests that the two tools optimise the hierarchy in different ways.

Table 4.6: Top MQ; significant results where Crunch values are higher are indicated in bold type

Case Study	Mean Top MQ			p-values		
	RMHC	GA	Bunch	Bunch/ RMHC	Bunch/ GA	RMHC/ GA
Module dependency graphs						
ttysnoop-0.12c	1.734	1.350	1.990	0.371	0.371	<0.001
compiler	**3.949**	**3.340**	1.700	<0.001	<0.001	0.002
mtunis	1.505	1.376	1.551	0.115	<0.001	0.024
ispell	**1.872**	1.695	1.652	0.005	0.538	0.039
rcs	**2.139**	1.942	1.750	<0.001	0.054	0.028
bison	**2.988**	**2.970**	1.174	<0.001	<0.001	0.924
dot	**2.452**	**2.391**	1.187	<0.001	<0.001	0.994
grappa	**7.129**	**5.557**	1.411	<0.001	<0.001	<0.001
aalib-1.3	**4.906**	**4.057**	1.192	<0.001	<0.001	<0.001
State machines						
liftController	0.955	0.958	1.000	0.001	<0.001	0.783
congestion	**1.166**	**1.265**	0.983	<0.001	<0.001	0.086
bgp	**1.519**	**1.449**	1.095	<0.001	<0.001	0.281
atm3	1.691	1.500	3.081	<0.001	<0.001	0.018
cvs	**2.913**	**2.468**	1.422	<0.001	<0.001	<0.001
roomcontroller	**2.687**	**2.580**	1.728	<0.001	<0.001	0.169
ssh	**2.062**	**1.807**	1.321	<0.001	<0.001	0.003
collections	**3.461**	**2.942**	1.297	<0.001	<0.001	<0.001
colt	**7.833**	**5.963**	1.345	<0.001	<0.001	<0.001

Table 4.6 summarises the MQ values of the top-most level in the hierarchy produced by each search type. Wilcoxon rank sum test p-values are reported in the p-value column; significant results (at $\alpha = 0.05$) are shown in bold type where the Crunch MQ value exceeded that of Bunch. In the majority of cases Crunch produced greater MQ values at this level than the corresponding Bunch modularisations, with the exception of the `ttysnoop`, `mtunis` and `atm3` case studies.

The MQ values for each of the lowest layers in the hierarchy are given in Table 4.7. It is at this level where Bunch has the greatest freedom to explore solutions and produces the smallest clusters and highest MQ values. While Crunch produced greater values at the top level, Bunch produced greater MQ values at this level in all cases except for the `congestion` and `bgp` case studies.

Table 4.7: Bottom MQ; significant results where Crunch values are higher are indicated in bold type

Case Study	Mean Bottom MQ			p-values		
	RMHC	GA	Bunch	Bunch/ RMHC	Bunch/ GA	RMHC/ GA
Module dependency graphs						
ttysnoop-0.12c	0.509	0.568	2.747	<0.001	<0.001	0.539
compiler	2.385	2.039	6.783	<0.001	<0.001	0.022
mtunis	1.228	1.288	2.990	<0.001	<0.001	0.842
ispell	1.540	1.031	3.289	<0.001	<0.001	<0.001
rcs	1.700	1.359	3.502	<0.001	<0.001	0.002
bison	2.257	1.809	4.825	<0.001	<0.001	0.005
dot	2.071	1.380	4.149	<0.001	<0.001	<0.001
grappa	4.682	3.378	13.370	<0.001	<0.001	<0.001
aalib-1.3	3.262	2.848	7.504	<0.001	<0.001	0.008
State machines						
liftController	0.866	0.872	1.000	<0.001	<0.001	0.931
congestion	**1.544**	**1.418**	0.983	<0.001	<0.001	0.023
bgp	**1.486**	**1.471**	1.095	<0.001	<0.001	0.929
atm3	0.894	0.949	3.081	<0.001	<0.001	0.666
cvs	2.022	2.205	3.881	<0.001	<0.001	0.134
roomcontroller	1.720	1.942	4.065	<0.001	<0.001	0.013
ssh	0.603	0.983	3.086	<0.001	<0.001	<0.001
collections	1.748	1.829	5.809	<0.001	<0.001	0.615
colt	4.229	4.058	12.069	<0.001	<0.001	0.318

The sum MQ results are shown in Table 4.8. These values estimate the overall quality of the hierarchy in terms of the optimisation produced. Values in this table show firstly that Bunch produced higher summed MQ values for all of the module dependency graphs. In the smaller state machine examples, Crunch produced higher values, while Bunch produced the greatest sum MQ values for the larger three state machines.

In all of the statistically significant pairs of GA and RMHC results for state machines, the RMHC algorithm produced higher mean sum MQ values than the GA. This trend is not observed in the MDGs, however. Fewer cases produced statistically significant differences in the RMHC and GA sum MQ values, although the GA produced greater mean values than the RMHC search for the larger dependency graphs.

Figure 4.14 shows the sum MQ values graphically in a box plot. The principal finding for this research question is that the hierarchies are structured differently by the two tools. Crunch appears to optimise MQ at the top of the hierarchy and tends towards the depth limit imposed by the fitness function for all case studies, while Bunch optimises the overall and bottom fitness values with depths varying between case studies.

Table 4.8: Sum MQ; significant results where Crunch values are higher are indicated in bold type

Case Study	Mean Sum MQ			p-values		
	RMHC	GA	Bunch	Bunch/ RMHC	Bunch/ GA	RMHC/ GA
Module dependency graphs						
ttysnoop-0.12c	4.485	3.921	4.274	0.059	0.050	0.001
compiler	9.711	9.821	11.741	<0.001	<0.001	0.684
mtunis	5.683	4.971	6.192	0.008	<0.001	<0.001
ispell	5.391	5.930	6.345	<0.001	0.002	<0.001
rcs	6.240	6.563	7.183	<0.001	0.006	0.080
bison	8.139	9.012	9.576	<0.001	0.059	0.005
dot	6.636	7.924	8.038	<0.001	0.819	<0.001
grappa	16.900	16.889	23.345	<0.001	<0.001	0.994
aalib-1.3	12.553	12.098	17.116	<0.001	<0.001	0.130
State machines						
liftController	**3.504**	**3.520**	1.467	<0.001	<0.001	0.970
congestion	**4.720**	**4.822**	1.522	<0.001	<0.001	0.739
bgp	**5.393**	**5.411**	2.470	<0.001	<0.001	0.641
atm3	**4.843**	**4.417**	3.500	<0.001	<0.001	<0.001
cvs	**8.602**	**7.830**	6.916	<0.001	<0.001	<0.001
roomcontroller	**7.976**	7.408	7.175	<0.001	0.144	0.001
ssh	5.519	4.673	6.146	<0.001	<0.001	<0.001
collections	9.016	8.315	12.274	<0.001	<0.001	<0.001
colt	18.609	16.892	26.222	<0.001	<0.001	<0.001

Discussion

The results at the top level show different fitness values between Crunch and Bunch, suggesting that the two tools do produce different structures in their solutions. This finding is corroborated by the reversal of superiority at the lower and overall levels, at which Bunch exceeded fitness values produced by Crunch in addition to the varying depth values produced by the two tools.

The poor performance at lower levels in Crunch could possibly be due to the depth limit imposed in the fitness function, which restricts the number of solutions it can use to represent a hierarchy. For example, it is possible to represent one cluster containing three nodes as a single cluster with three children, or a nested structure. Differences in the type of the hierarchy and the amount of nesting between Crunch and Bunch make the comparison of the two results difficult.

Depth values produced by Crunch were high; in almost all cases, it produced solutions with depths close to the imposed limit. Only in cases where Bunch produced fewer levels than Crunch did Crunch produce higher HMQ values. This suggests that the

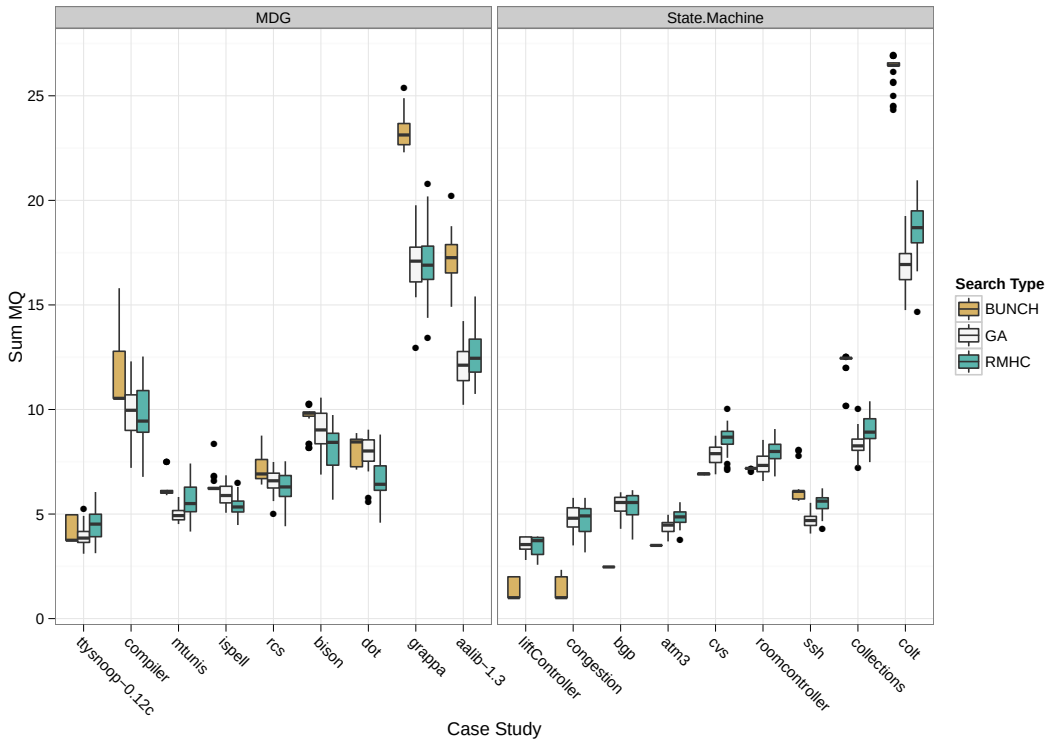


Figure 4.14: Box plot of sum MQ values produced

search may have been exploiting the duplication gained during the transformation to a layered hierarchy for assessment, and that the penalty term in HMQ does not successfully manage the depth of the hierarchies produced by Crunch. It also supports the hypothesis that direct comparison of the Bunch and Crunch hierarchies is not possible. The differences between the depths possibly indicates that the two tools are searching different parts of the search space.

Bunch produced the greatest MQ values at the bottom of the hierarchy, where it begins its optimisation. Crunch, however, does not have any such prioritisation, and its higher performance at the top levels suggests that it is easier for the search to produce arrangements that optimise the clusters higher up the tree than lower in the tree. The transformation procedure performed to enable the use of MQ with Crunch results may also have contributed to the poor performance, as layers of Crunch results are likely to contain clusters that contain very few elements. As MQ is defined as a ratio of intra-edges to inter-edges, these clusters may have had zero intra-edges and thus zero contribution to the MQ score at that level.

This study suggests that the more expressive representation used by Crunch incurs a cost of reduced MQ values, when compared to Bunch, using the HMQ fitness function. As MQ results for this fitness function were high at higher levels in the hierarchy, Crunch may require weighting by level or size for it to produce hierarchies more similar to Bunch. The comparison method at various layers is similar to the approach used by

Anquetil and Lethbridge [9] in their experiments on hierarchical clustering. As they remarked, selection of a cut-point is difficult; this is exacerbated by the difference in the hierarchies produced by each algorithm, and makes determination of a clearly superior algorithm in terms of hierarchical quality difficult.

Although the summed MQ values allow inferences about the ability of both searches to maximise MQ to be made, they do not assess the issue of quality. The appropriate amount of nesting is not known for any of the results, so it is not possible to assert that either tool produced better hierarchies overall. However, with respect to the research question of this study, the conclusion is that Crunch produces a higher top MQ value with a high depth, while Bunch maximises both the MQ of the bottom level and overall with fewer layers in the hierarchy.

Threats to Validity

The threats to validity for the results are as follows:

Results are from randomised algorithms and may not be representative This threat is mitigated by the 30 repetitions used for each of the search types. The likelihood the results are coincidence is low, as reported by the low p-values from the Wilcoxon rank sum test conducted for each pair of results. This test was used as it makes no assumption of the normality of the samples.

Bunch and Crunch levels may not correspond to one another As Crunch and Bunch formulate their hierarchies in different manners, the extraction of various layers exposes another threat to validity. This threat is intrinsic to MQ and can therefore not be removed from this experiment.

The MQ metric may not assess quality As with all metrics which serve to act as a surrogate of a subjective human, the use of MQ to assess the quality of the result means it is not possible to assert that the results are objectively more qualitative, only that the MQ values are better or worse. The purpose of this study was to compare MQ values, however, rather than the subjective quality of the results.

4.4.3 Conclusions of the Hierarchical Study

This experiment demonstrated that Crunch produces hierarchies that are largely different to Bunch. The higher MQ values at the top level, contrasting to the higher MQ values at the bottom level, suggests that Bunch may be more suitable if smaller modules at a lower level of abstraction is desired. The evaluation also indicated the difficulty in comparing the performance of the two hierarchical tools, irrespective of the structural differences that may or may not be apparent.

The difficulty in comparison provides evidence to suggest that HMQ and Crunch represent different priorities of optimisation compared to the agglomerative hill climbing approach used in Bunch. The next section describes an investigation into the structural differences of partitionings produced by different fitness functions implemented

in Crunch in order to further explore the nature of MQ to address the final threat to validity identified in the previous experiment.

§ 4.5 Alternative Fitness Functions

The previous chapter identified several issues with results from Bunch, those related to its hierarchy generation approach and those related to the structural quality of its output. The previous sections of this chapter explored the former problem through the use of Crunch with MQ. This section explores different fitness functions as a means to improve the quality of the result of a clustering operation for state machines when using the Crunch tool.

In this section, several alternative fitness functions are described and evaluated by using them to cluster two state machines with Crunch. The results are assessed from a structural standpoint, as well as through several fitness functions, some taken from existing approaches and others are original. These experiments are an extension of work presented by the author at ESEC/FSE 2011 [42].

These experiments address the research question "What types of structures are produced by Crunch using different fitness functions?".

4.5.1 Methodology

The objective of this assessment is to measure the relative performance of several fitness functions when used to cluster state machines in Crunch.

Two case studies, the largest state machines used in the previous studies, were clustered using both Crunch searchers (GA and RMHC) using each fitness function. The resultant MDGs were then assessed using several metrics to characterise the partitionings produced by each search.

Case Studies

Table 4.9 summarises the size of the case studies. The case studies were selected as they are the largest state machines used in the previous study; size was used as a selection criterion as larger graphs were more likely to exercise the hierarchy evaluation implemented in the hierarchical decomposition fitness function (described in Section 4.5.2). These state machines were also selected as they are reverse engineered from software, using the method described in Section 4.3.1, which makes them more realistic for the evaluation.

4.5.2 Fitness Functions

This evaluation surveys several fitness functions, each of which operate solely on the connectivity of the graph, and the partitions produced by the search algorithm. Each

Table 4.9: An overview of each case study.

Case Study	Transitions	States
Collections	38	26
Colt	63	53

of these fitness functions assess only the immediate level of the hierarchy, treating it as a partition, with the exception of the HMD fitness function which is a hierarchical measure.

XOR

Chu et al. [23] proposed an XOR-based heuristic to identify subsets of a state machine to be clustered in a superstate. This fitness function adapts their metric to the search-based approach used in Crunch. It does so by rewarding solutions that exhibit clusterings that comply to this XOR constraint. The more clusters doing so, the higher the fitness value. This is given by:

$$\sum_{C \in P} \sum_{x \in C} \frac{|\epsilon_x|}{|\epsilon_C|}$$

where $P = \{C_1, C_2, \ldots C_n\}$ is a partition of the set of nodes. For a graph $G = (V, E)$, a node $x \in V$ in the cluster C_i has inter-edges $\epsilon_x = \{e | (x, v) \in E \wedge v \notin C_i\}$. The inter-edges of a cluster C_i is the union of all inter-edges e_x of each node within it: $\epsilon_{C_i} = \bigcup_{x \in C_i} \epsilon_x$. Clusters containing 0 or 1 nodes are given a fitness value of 0.

Cyclomatic

A cyclomatic complexity (CC) fitness function; given by:

$$\frac{1}{max(E - N + 2, \texttt{Double.MIN\_VALUE})}$$

This can be seen to approximate Kumar's method [62] of evaluating statechart inference from state machines. A transformation is applied that removes transitions occurring from every state in a superstate, replacing them with a single transition from the superstate. The superstates themselves are ignored in this calculation, such that N never changes, although E decreases when this rewriting occurs. The reciprocal is taken such that reducing CC increases the fitness of the result.

Edge Reduction

This fitness function operates on a similar principle to the cyclomatic fitness function but instead directly represents the fitness by the number of edges which can be removed by introducing superstate transitions to the hierarchy. This models the approach to composite state generation described by Systä et al. [108] in a search-based context.

HMD

An implementation of Lutz's HMD [69] fitness function. Based on the minimum description length principle from Information Theory. It estimates the number of bits required to express a given clustered state machine using a formula; the less complex the hierarchical machine, the lower the number of bits required to express it.

One component of the encoding is the set of transitions. The number of bits required to represent transitions is reduced when the start and end states appear closer in the hierarchy. This is balanced against the number of clusters, each of which increases the total number of bits required to represent the hierarchy.

The reciprocal of the number of bits required to estimate the hierarchy is taken, such that solutions requiring fewer bits receive higher fitness values [69].

Edge Similarity

This fitness function maps each state to a coordinate based on its connectivity to other states, similar to van Deursen and Kuipers' use of the vector space model for software clustering [118]. The coordinates of a state s are given as a vector of 1s and 0s which denote the existence or non-existence of a transition from s to each $t \in N$.

The similarity of two states s and t is defined as the euclidean distance between them. This is given by:

$$dist = \sqrt{\sum_{i=0}^{n} (u[i] - v[i])^2}$$

where u and v represent the coordinates of states s and t respectively.

The similarity between two modules is defined as follows:

$$moduleFitness = \begin{cases} \frac{1}{1+internal} & \text{if } external = 0 \\ \frac{external}{external+internal} & \text{otherwise} \end{cases} \tag{4.1}$$

Where *external* is the sum the distance between each state within the cluster and all states outside it and *internal* is the summed distance between each of the states within the cluster. The mapping to an ordinal scale used based on the mapping of coupling levels to ordinals used by Fenton and Melton [35].

The calculation is performed for each module at the top level of the hierarchy and summed to produce the final fitness measure for the modularisation.

This fitness function aims to increase the distance of elements in the cluster with those outside it, but also rewards assignments with a low internal distance. Solutions that have a higher separation of external nodes are more permissive of internal dissimilarity.

MQ

The original TurboMQ fitness function implemented in Bunch. This fitness function allows the comparison of the performance of other functions against MQ without considering the search method of each tool. This function calculates fitness only at the top level of the hierarchy.

4.5.3 Search Configuration

Both the genetic algorithm and random mutation hill climbing searchers from Crunch were used. The parameters to the GA were kept at the JGAP default, with a population size of 50. Both search algorithms were assigned a fitness evaluation budget of $50,000$. As Crunch uses randomised algorithms, 30 repetitions were conducted for each fitness function, search type and case study, resulting in a total of $6 * 2 * 2 = 720$ clusterings.

4.5.4 Metrics

Assessment of the results is conducted at the top level of the tree, as all but one of the fitness functions use only this level for fitness evaluation. This level contains the largest clusters in the tree, and is obtained by selecting the head node of the tree's immediate children. The following metrics for each of the top-level partitionings are used:

MQ The MQ value at the top level of the hierarchy

Number of inter-edges The number of transitions that leave or enter another cluster

Edge Reduction The number of edges that can be reduced according to the composite state rule used by Systä et al. [108] and Kumar [62]

Number of clusters The number of top-level clusters in the output

4.5.5 Results

Table 4.10 summarises each of the metric values for each of the sets of 30 clustered MDGs. The rows headed "RMHC" and "GA" are results from the random mutation hill climber and genetic algorithm respectively.

The results firstly show that several of the fitness functions failed to produce a reduction in the number of inter-edges; mean values for the Cyclomatic, Edge count reduction and Edge Similarity fitness functions were all close to the number of edges in the machine for both case studies (there was one reflexive edge in each of the state machines that could not be clustered to turn it into an inter-edge). The XOR fitness function performed similarly, although it produced fewer inter-edges than these fitness functions on average. Both HMD and Bunch MQ produced a higher decrease in the number of inter-edges.

Comparing the numbers of clusters, Edge Similarity produced the largest mean number of clusters, the largest of all the fitness functions for both cases, in addition to producing

Table 4.10: Summary statistics for the fitness function experiment

		Fitness Function	MQ				Num. Clusters				Inter-Edges				Edge Reduction			
			Mean	Min	Max	SD	Mean	Min	Max	SD	Mean	Min	Max	SD	Mean	Min	Max	SD
Collections	GA	BunchMQ	4.676	3.417	5.429	0.539	13.400	11.000	17.000	1.404	23.200	20	27	1.919	0.000	0.000	0.000	0.000
		Cyclomatic	0.616	0.286	1.341	0.270	15.967	13.000	19.000	1.691	34.233	32	36	1.040	5.133	4.000	6.000	0.900
		EdgeReduction	0.632	0.286	1.291	0.284	16.267	12.000	19.000	2.067	34.467	33	36	0.860	5.167	2.000	6.000	1.177
		EdgeSimilarity	0.290	0.200	0.750	0.134	24.967	24.000	26.000	0.320	36.900	36	37	0.305	0.000	0.000	0.000	0.000
		HMD	2.153	1.059	3.262	0.509	8.967	6.000	14.000	1.564	22.733	9	30	4.291	0.067	0.000	2.000	0.365
		XOR	1.220	0.382	2.082	0.436	12.933	9.000	17.000	2.067	33.100	29	35	1.539	0.667	0.000	3.000	1.061
	RMHC	BunchMQ	4.579	2.750	5.529	0.622	13.267	8.000	16.000	1.893	23.367	15	28	2.710	0.000	0.000	0.000	0.000
		Cyclomatic	0.605	0.167	1.167	0.293	16.567	13.000	21.000	2.012	34.200	30	37	1.789	3.467	0.000	6.000	1.995
		EdgeReduction	0.630	0.250	1.345	0.268	15.733	11.000	18.000	1.530	34.133	27	37	1.889	3.267	0.000	6.000	2.067
		EdgeSimilarity	0.401	0.111	1.417	0.275	22.333	20.000	25.000	1.398	36.400	34	37	0.814	0.067	0.000	2.000	0.365
		HMD	2.553	1.183	4.207	0.654	6.200	3.000	10.000	1.424	16.833	6	23	3.563	0.000	0.000	0.000	0.000
		XOR	1.160	0.200	2.308	0.540	13.833	10.000	19.000	1.931	33.867	29	37	1.995	0.433	0.000	3.000	0.898
Colt	GA	BunchMQ	8.809	7.067	10.333	0.729	28.500	25.000	34.000	2.570	38.700	34	43	2.184	0.000	0.000	0.000	0.000
		Cyclomatic	0.488	0.125	1.778	0.401	32.633	28.000	38.000	2.773	60.267	57	62	1.112	2.433	1.000	3.000	0.626
		EdgeReduction	0.555	0.125	1.347	0.354	31.900	24.000	38.000	3.166	59.833	55	62	1.464	2.267	1.000	3.000	0.640
		EdgeSimilarity	0.550	0.333	1.333	0.215	50.133	48.000	52.000	0.937	61.833	60	62	0.461	0.000	0.000	0.000	0.000
		HMD	4.084	2.411	6.006	0.736	18.033	13.000	24.000	2.428	40.067	34	49	3.331	0.000	0.000	0.000	0.000
		XOR	1.776	0.417	3.976	0.981	27.667	24.000	35.000	3.122	57.667	54	61	1.936	0.267	0.000	2.000	0.521
	RMHC	BunchMQ	9.525	8.202	11.111	0.751	27.100	22.000	33.000	2.383	37.467	32	43	2.270	0.000	0.000	0.000	0.000
		Cyclomatic	0.521	0.182	1.293	0.313	32.867	24.000	38.000	3.309	60.267	57	62	1.258	1.333	0.000	3.000	0.884
		EdgeReduction	0.611	0.100	1.243	0.294	31.700	27.000	38.000	3.007	59.633	56	62	1.351	1.367	0.000	3.000	0.928
		EdgeSimilarity	0.493	0.250	1.000	0.156	45.733	43.000	48.000	1.552	61.833	61	62	0.379	0.000	0.000	0.000	0.000
		HMD	3.700	1.710	6.211	1.034	8.300	5.000	11.000	1.466	20.200	9	28	4.302	0.000	0.000	0.000	0.000
		XOR	2.100	0.833	4.167	0.833	28.133	22.000	34.000	3.126	57.167	53	61	1.859	0.167	0.000	1.000	0.379

the largest mean number of inter-edges. The HMD fitness function produced the lowest mean number of clusters in all cases.

Edge reduction means for the majority of the fitness functions were zero, although in some cases the XOR metric produced arrangements that allowed superstate transitions to reduce the number of edges in the machine. The greatest reduction of edges was produced by the two fitness functions that use it as an optimisation criterion: Cyclomatic and Edge Count Reduction.

The MQ values produced by each fitness function show that, while MQ optimises it best (a result to be expected), Both HMD and XOR appear to produce increases in MQ, when compared to the remaining fitness functions. Figure 4.15 shows a box plot of the MQ values each search and fitness function produced for both case studies. The whiskers extend to the highest and lowest values within 1.5 times the inter-quartile range, outliers are drawn as points.

With respect to the research question, the experiment has determined that HMD produces larger clusters and also appears to correlate with MQ more-so than the superstate transition-based fitness functions, although HMD and MQ did not enable any superstate transitions to be rewritten in most cases.

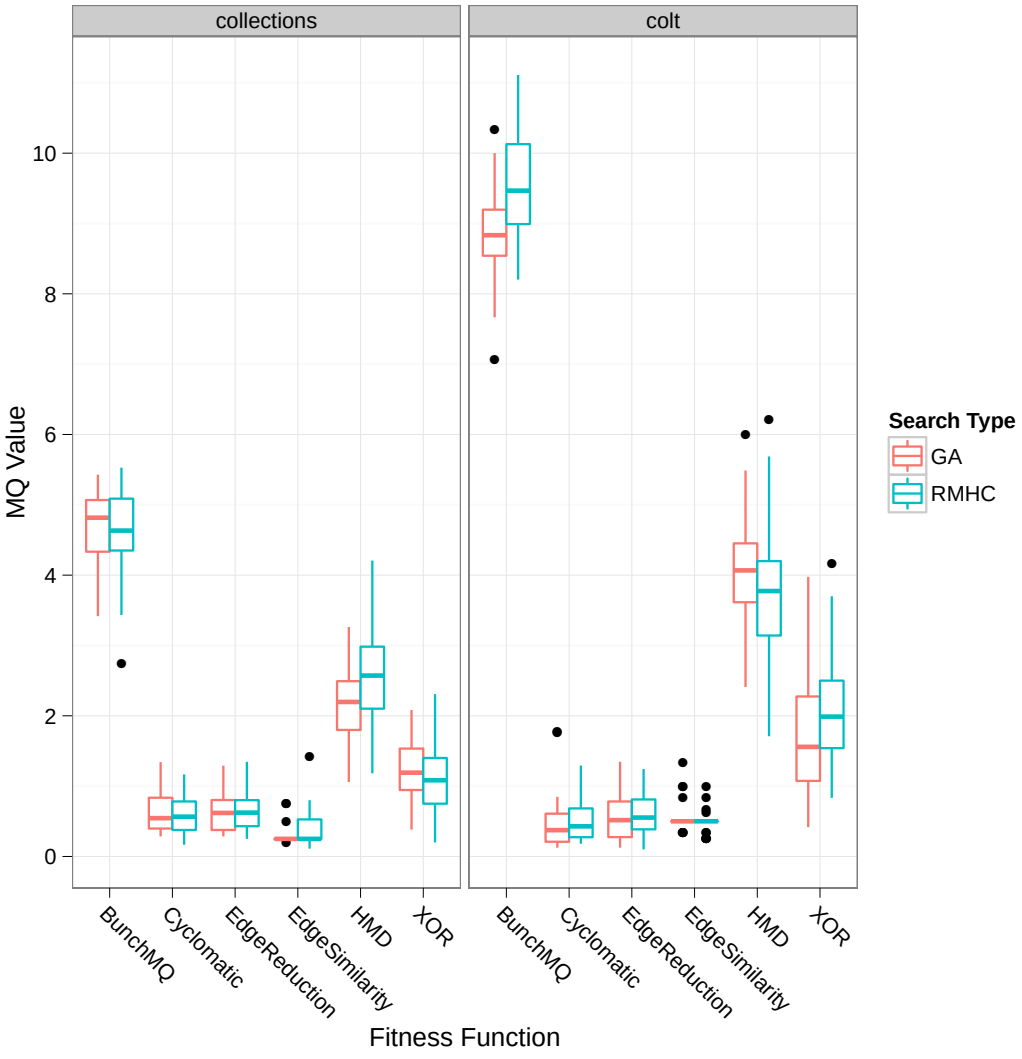


Figure 4.15: Box plot of MQ values produced by search type

4.5.6 Discussion

Both the Cyclomatic Complexity fitness function and Edge Reduction metrics produce highly similar results. Pairwise Wilcoxon tests produced p-values for each of the metrics suggest that the null hypothesis, that the two values are drawn from the same population, cannot be rejected as they were all considerably greater than the 0.05 significance level. This suggests that cyclomatic complexity can be replaced with a single edge reduction without any change in the performance of the result.

The poor performance of the Edge Similarity, Cyclomatic Complexity and Edge Reduction fitness functions in MQ is possibly due to their poor performance at reducing the number of inter-edges. Although these fitness functions produced arrangements that allowed transitions to be rewritten as composite transitions, they did so at a cost of increased numbers of inter-edges.

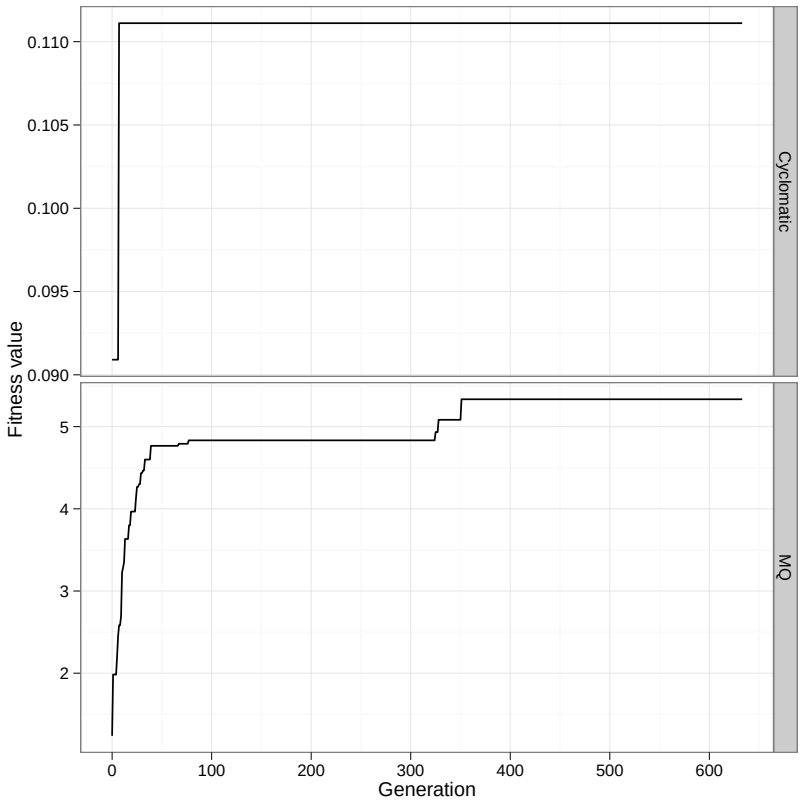


Figure 4.16: Change in fitness over evaluations for MQ and Cyclomatic fitness functions

Figure 4.16 shows a line plot of the maximum fitness value of the populations for two genetic algorithm searches for the collections case study, for the MQ and Cyclomatic Complexity fitness functions. This figure illustrates a possible problem with the edge reduction based metrics; the fitness landscape is likely to be of "stepped" construction. Both Cyclomatic Complexity and the Edge Reduction metrics do not offer a means to

represent a partial improvement, which reduces the tolerance of the fitness functions to local optima. This demonstrates that an ideal fitness function for clustering must encode a partial improvement in its fitness function.

The performance of HMD appears to replicate Lutz's findings, in that the HMD fitness function was able to produce higher MQ values than the other fitness functions (excluding MQ itself) [69]. HMD also produced the fewest clusters of all the fitness functions; this is likely due to the hierarchical optimisation it performs. As the other fitness functions only optimise at the level assessed by the metrics, the large clusters observed for HMD may be due to the optimisation occurring at lower levels.

The XOR fitness function also produced higher MQ values than the remaining three fitness functions (edge similarity, Cyclomatic Complexity and edge reduction), in addition to locating several solutions that allowed composite transitions to replace a small number of edges.

It is not possible to conclude which of the two search techniques best optimises the MQ metric, as evidenced by the box plot in Figure 4.15. Although in some cases RMHC produces greater maximum values, there is not a clearly superior technique in any of the cases.

Although the quality cannot be measured through these results, the data still shows that the superstate identification problem described by Systä et al. [108] may be solved using a search-based approach. However, comparing the MQ values against the number of edge reductions suggests that the quality may suffer where the number of superstate transitions is maximised.

Threats to Validity

There are several factors which may pose as threats to the validity of the findings in this study:

Results may be due to chance The use of 30 repetitions for each search type, and the use of the non-parametric pairwise Wilcoxon tests with Holm-Bonferroni p-value adjustment ensure that the comparisons made in this experiment are unlikely to be due to chance.

Metrics do not assess quality Quality is subjective, and as such it is not possible to interpret these metrics as indicators of quality, beyond the objective criterion of superstate transition identification and the evidence in the literature that MQ correlates with quality. However, the objective of this experiment was to characterise the type of hierarchy produced by these fitness functions and not to determine which was more qualitative.

HMD is hierarchical while the other fitness functions are not This fitness function optimises a hierarchy, rather than only the level investigated by these metrics. As a result, the larger clusters it produced may be due to the lower-level structures it has produced. While this impacts the interpretation of quality of the

result, it does not forbid analysis of the structures produced by the fitness function, although the comparison cannot be considered to operate at the same level for HMD as the other fitness functions.

4.5.7 Conclusions of the Empirical Study

This study has demonstrated how several metrics described in the literature can be applied as fitness functions in search based clustering. The evaluation of these measures showed that the modularisation metrics (MQ and HMD) did not produce hierarchies that enabled superstate transitions to be produced. In contrast, the fitness functions that aim to optimise the clusters to enable superstate transitions to replace several transitions did so at a cost of MQ values and high numbers of inter-edges.

Difficulties in selecting an adequate measure were demonstrated by the apparent conflict of these two concerns. This suggests that the problem of state machine clustering cannot be reduced to a single objective function value. In addition, this comparison was only performed at one level of a hierarchy. The comparison results in the previous experiment demonstrated the difficulty of evaluating such hierarchies, which further complicates the derivation of a metric for state machine clustering.

Although the study did not produce a single objectively superior metric of those evaluated, it offers the insight that in order to solve the superstate identification problem with the goal of reducing edges via superstate transitions, the fitness function must encode partial quality. This was evidenced by the lack of continuous improvement observed for the Edge Reduction and Cyclomatic Complexity metrics.

§ 4.6 Conclusions

This chapter introduced Crunch, a novel search-based clustering tool that avoids the problems in the Bunch approach to hierarchy generation identified in Chapter 3 using a genetic programming approach. This approach allows whole hierarchies to be searched at once, rather than level-by level, thus avoiding over-fitting and the constraint to layered hierarchies.

Crunch was evaluated in three aspects in this chapter. Firstly, in Section 4.3, comparison at the single level showed that the hierarchy representation produces lower fitness values than Bunch when evaluated only at one level. This demonstrated that Crunch's approach to searching whole hierarchies negatively impacts its ability to optimise a single-level hierarchy. The various representations and search types described in this chapter were also compared, showing that of the genetic programming implementations, the symbol gene produced the fittest solutions.

The second evaluation measured the performance of Crunch and Bunch for the hierarchy generation task in Section 4.4.2. Assessment of the MQ values at various levels showed that Crunch can produce higher fitness values at the top levels of the hierarchy, at a cost of lower fitness values further down the hierarchy, when compared to Bunch.

On comparing several fitness functions, the HMD [69] fitness function reduced the number of inter-edges and clusters the most, although it did not identify as many candidate superstate transitions. The results from this experiment firstly capture the difficulty in defining fitness functions and assessment criteria for clustering. While MQ has been found to be a good estimator of modularisation quality, like all metrics it serves as a "rule of thumb" and care must be taken to not lend too much authority to it. The reported negative industry experience with the use of Bunch [40] gives credence to this observation. Additionally, the evaluation shows that assessment relies on many variables, suggesting that production of a single fitness function to solve the problem is impractical, especially when the subjectivity of interpretation of the metrics is considered; a reduction in inter-edges or clusters is not objectively good or bad, judgement by a domain is necessary.

Mechanised remodularisation algorithms are therefore not a panacea. Although efforts may be directed at improving metrics to increase authoritativeness, the subjectivity of the modularisation problem cannot be ignored, as it is an inescapable part challenge, besides identification of structure according to design quality. With these facets considered, it is then reasonable to attempt to remove subjectivity from the challenge by introducing a human into the remodularisation process. The following chapters examine this, using an interactive approach to defer the subjective judgement to a human expert, while leaving the non-subjective component to automated approaches.

Chapter 5

SUMO: Supervised Software Remodularisation

As shown in Chapters 3 and 4, automated remodularisation can be used to cluster state machines, however the results may not always match a domain expert's intuitions. This mismatch can arise if the developer makes subjective decisions based on additional knowledge, beyond the information present in the structure of the graph. In this circumstance, the remodularisation algorithm, operating on a partial set of the required knowledge may not be able to produce as good a result. This chapter addresses this problem using an interactive approach of soliciting additional domain knowledge from an expert in order to produce a clustering which incorporates this domain knowledge to better match the expert's intuitions.

The SUMO approach described in this chapter is motivated by the body of work that uses domain knowledge to improve remodularisation [12, 74, 90, 100] as well as clustering [120]. These approaches, discussed in detail in Sections 2.5.3 and 2.5.2, use human intuition to assist the algorithm in order to increase the quality of the result. This chapter applies a *refinement* process that uses constraints to correct erroneous elements of a modularisation identified by an expert in a similar manner to the constrained clustering approach used by Wagstaff et al. [120].

Expert time is costly, so the refinement process must be as efficient as possible, in terms of the improvement produced as a result the expert's supplied information. This chapter introduces SUMO, an algorithm that uses pair-wise constraints as domain knowledge to achieve this. The amount of domain knowledge needed to produce a full refinement is quantified both in theoretical terms and though an evaluation of the algorithm, using 150 simulated authoritative decompositions generated from five software systems. The evaluation shows that while a large amount of information is required to completely refine a Bunch-produced modularisation, incremental improvement can be gained within a short number of iterations using the SUMO refinement technique.

This chapter makes the following contributions:

1 The SUMO constraint-based refinement algorithm that allows a domain expert to interactively refine a modularisation, including equations for the minimum amount of information required to produce an unambiguous result.

2 An empirical evaluation of SUMO that uses simulated authoritative decompositions from five diverse software systems as the source of domain knowledge, which finds that SUMO is able to produce regular partial improvements to the modularisation in all cases.

§ 5.1 Refining Modularisations

The previous chapters showed that automated remodularisation techniques can be applied to produce hierarchies for state machines. However, the focus of these chapters was on the applicability of the approach (in Chapter 3) and the performance of the Crunch clustering tool in terms of MQ values, and the structures it produces (in Chapter 4). Beyond the initial investigation into the use of Bunch on state machines, the issue of the accuracy has not been addressed thus far.

Although Chapter 4 showed that Crunch can produce higher MQ values at the top level of a hierarchy than Bunch, it did not show that these higher MQ values were more qualitative. This criticism also stands for Bunch; MQ requires validation in order to determine how suitably it approximates the quality of a modularisation. However, both Bunch and Crunch operate on a fraction of the information available for the system that is represented in the graphs they operate on.

Part of the remodularisation problem is in the handling of subjective decisions that form part of the evaluation of a result. Automated approaches can only do their best to approximate these subjective decisions based on the available indicators, however. In a survey of six automated remodularisation algorithms, Wu et al. [126] found that similarity to the known-good modularisations for several versions of five software systems was low.

Glorie et al. [40] reported that the application of Bunch to an industrial problem was unsuccessful. Although Bunch provides several features that allow the user to direct it, the difficulty in judging the result, and comparison of different results produced by Bunch made the process more challenging. Domain experts in their study found that solutions produced using Bunch were “non-acceptable”. Although Bunch provides a means to start the modularisation process from a solution, Glorie et al. [40] found that the practical application of this seeding of the search was difficult as Bunch may modify some of the user-supplied modules. Following from this, an automated technique must allow the user to supply their domain knowledge in the simplest way possible.

These problems suggest that a means to interactively incorporate domain knowledge may be used to improve on existing remodularisation techniques. Wagstaff et al. [120], applied a constraint solver to the k-means clustering algorithm to improve the quality

of the clusterings produced by it. SUMO applies a similar approach, although it is a post-hoc refinement step, rather than an interactive clustering algorithm. The following section describes the SUMO algorithm. This use of a refinement step allows the choice of the remodularisation algorithm to be made based on the scenario, before refinement is used to hone it into an acceptable result.

§ 5.2 Interactively Refining Modularisations

SUMO uses pair-wise constraints to refine a modularisation. These constraints are solicited as corrections to a hypothesis modularisation, supplied by a user. The objective of the user is to supply a sufficient set of pairwise constraints to transform the existing solution into the desired modularisation. As the user is surveying pairs of items, the refinement occurs piece-wise, such partial refinement therefore is possible using the algorithm.

The following section describes the SUMO algorithm, and shows how a constraint solver can be used to produce modularisations that conform to the user's supplied domain knowledge. A running example is used to illustrate the various features of the SUMO algorithm and is detailed in the following subsection.

5.2.1 Motivating Example

In this example, a module dependency graph of a small toy system will be used. The toy system contains 9 classes, three of which are associated with a Model View Controller architecture, and the remainder are associated with I/O and handling of XML data. Figure 5.1 shows the dependency graph of the system, as clustered by Bunch.

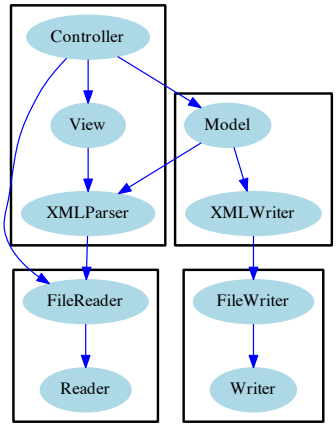


Figure 5.1: The example MDG, clustered by Bunch

In this example, suppose that the required set of packages is as follows:

Model	Model
View	View
Controller	Controller
I/O	FileReader, FileWriter, Reader, Writer
XML	XMLParser, XMLWriter

In a remodularisation scenario, this set of packages would not be available, and a tool such as Bunch would be relied upon to produce the solution. There is a difference between these solutions however. While Bunch grouped elements based entirely on their dependencies, the expert decomposition incorporates additional domain knowledge that is not encoded in the structure of the graph alone.

This is the motivating scenario for the use of refinement as implemented in the SUMO algorithm. Although this toy system is sufficiently small to enable remodularisation to be performed manually, a realistic remodularisation scenario may entail significantly more classes, for instance the system studied by Glorie et al. contained approximately $30,000$ source files [40]. The small example is used to illustrate elements of the SUMO algorithm, however it may be applied to larger systems, and as will be shown in the evaluation in Section 5.4 it is possible to use SUMO to produce incremental improvements to larger case studies through its interactive approach to the problem.

5.2.2 The SUMO Algorithm

The objective of a refinement process for remodularisation is to transform the existing modularisation, the hypothesis, to produce a result that is better in the sense that it more closely matches a user's intuitions. In order to achieve this, SUMO encodes these intuitions as pair-wise constraints between elements, which may be classes or states in MDG or state machines respectively. Wagstaff et al.'s must-link and cannot-link relation types [120] are used, termed positive and negative in SUMO.

SUMO allows the modularisation to be reasoned about in terms of these relations. This allows the user to focus on parts of the modularisation, rather than reorganising it entirely in one step. In the motivating example described in Section 5.2.1, a developer, on assessing the output of Bunch, may identify a correction that could be applied. For example, the FileWriter and FileReader classes should be placed together and the Model, View and Controller classes should appear separately. SUMO allows this information to be incorporated interactively to allow the modularisation to be refined as it is explored.

SUMO operates on the set of domain knowledge provided by the user, stored as unordered pairs of elements in the R^+ and R^- sets respectively. Let C be the set of all n entities in the modularisation (classes or states) $\{c_1 \dots c_n\}$. M is the set of modules, $\{1 \dots m\}$ where m is the number of modules and $m \leq n$. A solution is a mapping $a : C \to M$ that assigns a module to each of the elements in the input.

For example, to encode the positive relation between FileWriter and FileReader classes, that they should be together, the pair $\{FileWriter, FileReader\}$ would be placed in the R^+ set. Each pair of the Model, View and Controller classes would be placed in the R^- set to encode that they should not be together. The advantage of SUMO is that these constraints can be supplied interactively, as necessary by the expert.

Algorithm 6 The SUMO algorithm

```
R+ ← ∅
R− ← ∅
a ← inputModularisation
repeat
    showHypothesis(a)                          ▷ Elaborated in Section 5.2.2
    R+′ ← R+
    R−′ ← R−
    (R+, R−) ← addConstraints(R+, R−)          ▷ Elaborated in Section 5.2.2
    if constraintsViolated(a) then             ▷ Elaborated in Section 5.2.2
        R+ ← R+′
        R− ← R−′
        showConstraintViolatedError()
    end if
    a ← getHypothesis(R+, R−)                  ▷ Elaborated in Section 5.2.2
until userIsSatisfied
return a
```

Algorithm 6 summarises the SUMO algorithm. The process begins with an existing modularisation, which may be from a remodularisation tool such as Bunch (as used in the motivating example) or an existing package hierarchy. The expert repeats a process of analysing the algorithm's hypothesis and supplying corrections in the form of constraints where necessary, until a termination condition is reached. The steps are elaborated below.

showHypothesis

This function presents a hypothesis solution a to the user. The user uses their intuition to identify elements of the modularisation that conform to their domain knowledge, or violate it. The user also has the opportunity to examine the state of the solution at each step so progress can be visualised.

addConstraints

At this step, the feedback identified by the user is imported into SUMO's R^+ and R^- sets in the form of pair-wise constraints. If the user has identified that two elements should be clustered together or apart, the pair of elements is placed into R^+ and R^- respectively. This reduces the clustering problem from a subset identification problem to individual judgement of pairs of elements. If the user supplies new constraints that

violate previous constraints, the copies of the R^+ and R^- sets are restored to allow the process to continue without this inconsistency.

The pairs of information are not necessarily selected individually the user. In the motivating example, to encode that the Model, View and Controller classes should all be separate requires that a negative relation be added for all pairs containing each of these elements. A method to perform this step automatically at the user's direction to reduce the effort of encoding such solutions is implemented in the later version of SUMO built into a GUI, described in Section 6.2.

constraintsViolated

A set of corrections may violate constraints already provided. For example, if a user identifies that pairs of elements a, b and b, c should be together, i.e. $R^+ = \{\{a, b\}, \{b, c\}\}$. In this case it is possible to deduce that all three elements should be together. If the expert later determines that b and c should be separate, the pair is added to R^-. However, in this example scenario, given the set of constraints already supplied, it is not possible to assign b and c to another module; there is a conflict and the set of constraints is not solvable.

If such a configuration where the user contradicts previous corrections emerges, an error message is presented and the most recently added corrections are dropped. This is not the only way such a conflict could be handled, however. For example, the user could instead be shown the set of conflicting constraints and invited to choose the one which is most appropriate.

getHypothesis

A new solution is generated which conforms to the new, expanded set of constraints in the R^+ and R^- sets. This solution is generated using the CHOCO [55] constraint solver, which searches for a set of assignments a which conforms to each of the supplied constraints. The mapping of the problem to a constraint satisfaction problem is further described in Section 5.2.3.

Termination

The algorithm completes when the user indicates that the modularisation has been suitably refined, i.e. when they provide no more corrections. This encompasses various criteria, including the quality of the result: the budget (in terms of time the user has to perform the refinement) may also be a constraint. As the user is responsible for termination, the process can end before the set of constraints unambiguously produces the same solution. The bounds of the sizes of the sets R^+ and R^- are explored later in Section 5.3. The following section explains how the set of constraints can be translated into a constraint satisfaction for use with a constraint solver.

5.2.3 Modularisation as a Constraint Satisfaction Problem

The problem is mapped to a constraint satisfaction problem (CSP) $\langle X, D, C\rangle$ [99, p. 137] as follows:

The set of variables X represents each class, such that $|X| = n$ for n classes.

The domain D is the "module number" of the class, $D = \{m : m \in \mathbb{N}, 1 \leq m \leq n\}$. The problem is then to produce the mapping $a : X \to D$, where the module of a class $c \in X$ is given by $a(c)$. The mapping of a hypothesis solution a is the same as in Algorithm 6, it assigns each class to a module.

The set of constraints C consisting of: a pair of variables $v1, v2 \in X$ and a relation, one of $\{=, \neq\}$.

As an example, let

$$\begin{aligned} C &= \{XMLParser, Parser, XMLWriter\} \\ R^{+} &= \{\{XMLParser, Parser\}\} \\ R^{-} &= \{\{XMLParser, XMLWriter\}\} \end{aligned}$$

The mapping to a CSP results in:

$$\begin{aligned} X &= \{XMLParser, Parser, XMLWriter\} \\ D &= \{1, 2, 3\} \\ C &= \{\langle(XMLParser, Parser), =\rangle, \langle(XMLParser, XMLWriter), \neq\rangle\} \end{aligned}$$

Before any relations are applied, the constraint solver has to identify a solution to the following set of assignments:

```
XMLParser = [1,2,3]
Parser    = [1,2,3]
XMLWriter = [1,2,3]
```

Having been supplied the example assignments, the constraints can only be met if both $XMLParser$ and $Parser$ are not equal to $XMLWriter$, i.e. the solution will be $\{XMLParser, Parser\}, \{XMLWriter\}$, one of the possible encodings of this solution is $XMLParser = 1, Parser = 1, XMLWriter = 2$.

The remainder of the chapter evaluates the SUMO algorithm, starting with quantification of the sizes of the R^{+} and R^{-} sets.

§ 5.3 Sizes of the R^+ and R^- Sets

The performance of the algorithm is first quantified in theoretical terms. With the definitions given in Section 5.2.2, the total set of all positive relations for a given modularisation a is the set of all unordered pairs of classes in C that are placed in the same module:

$$R^+ = \{\{c_i, c_j\} : c_i, c_j \in C, a(c_i) = a(c_j), c_i \neq c_j\} \tag{5.1}$$

Similarly, the set of negative constraints R^- is defined as:

$$R^- = \{\{c_i, c_j\} : c_i, c_j \in C, a(c_i) \neq a(c_j), c_i \neq c_j\} \tag{5.2}$$

The total set of domain knowledge is then the set of all pair constraints that have been supplied, positive and negative: $R^+ \cup R^-$. This serves as an estimate of the amount of effort invested into the refinement process; the larger the union of these sets, the more knowledge has been transcribed to constraints of pairs.

The following subsections quantify the maximum and minimum size of the set R for the most and least general modularisations of a modularisation to be refined respectively. These informal proofs assume that all assignments in the function a are made according to relations supplied by a user i.e. a does not encode "guesses" by the solver. The impact of this assumption is that the starting condition is irrelevant: if only supplied constraints are used there are no possible solutions until R contains sufficient information. Following from this quantification of the upper and lower bounds on required relations, a general value for the size of R is then derived for arbitrary numbers of classes n and modules m.

5.3.1 Worst-case

The worst-case informal proof presented in this section was contributed by Neil Walkinshaw in the conference paper written as a result of the research presented in this chapter [43].

In the worst-case scenario, the target modularisation places each element in its own cluster, i.e. the assignment mapping is the identity function, $a(i) = i$. In this scenario, there are no transitive properties to exploit; negative information cannot be used to infer relations like positive information can.

The set R of relations will therefore consist entirely of negative relations:

$$\begin{aligned} R^+ &= \emptyset \\ R &= R^+ \cup R^- \\ |R| &= |R^-| \end{aligned}$$

By definition 5.2, R^- is the set of all unordered pairs of modules such that $a(c_i) \neq a(c_j)$. As in this scenario this inequality is always satisfied ($a(i) \neq j : \forall j \in M \setminus \{i\}$), the size of R^- is given by the number of distinct pairs of elements of the set C, i.e. $|R^-| = |R^|| = \frac{n(n-1)}{2}$.

5.3.2 Best-case

In the best case scenario, there is only one module, the relations are all positive, and therefore transitive. This equation and the subsequent general-case equation are original and were derived by the author.

Let $G = (V, E)$ be an undirected acyclic graph. The set of vertices V is given by $V = C$ and the number of classes n, is $n = |C|$. The set of edges is given by $E = R$.

Let $R^- = \emptyset \Rightarrow E = R^+ \cup \emptyset$. As $R^- = \emptyset$, the solution is reached when all classes are grouped in the same module, i.e. $a(c_1) = a(c_2) : \forall c_1, c_2 \in C$.

This solution requires R^+ to contain a relation for each class, such that there exists a path in G between any $v1, v2 \in V$. $|E|$ (and therefore $|R^+|$) is then the minimum number of edges required to connect G such that G is a tree. For this to be the case $|E| = |V| - 1$, therefore $|R| = |R^+| = n - 1$.

5.3.3 General Case

Through the definition in the previous proof, each distinct module is a connected component in the graph. $|R^+|$ is therefore $n - m$ where m is the number of modules, as increasing m by 1 consists of deleting 1 edge from the graph to split a connected component in two.

One negative relation must exist between each component, so there must be at least $\frac{m(m-1)}{2}$ negative relations (assuming the minimum set of positive relations is given).

The number of relations required can then be generalised:

$$
\begin{aligned}
|R^+| &= n - m && (5.3)\\
|R^-| &= \frac{m(m-1)}{2} && (5.4)\\
|R| &= |R^+| + |R^-| && (5.5)\\
&= n - m + \frac{m(m-1)}{2} && (5.6)\\
&= \frac{m(m-3)}{2} + n && (5.7)
\end{aligned}
$$

The performance of SUMO is therefore dependent on the size of the MDG n and the number of modules in the target mapping m. As m is the most significant term in the expression for $|R|$, the less general the target modularisation (higher values of m), the more relations that will be required. A more general result will require fewer constraints due to the transitive property of the positive relation type.

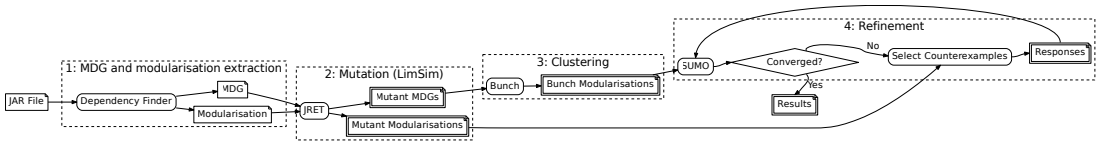


Figure 5.2: The process of evaluating SUMO on a case study.

§ 5.4 Empirical Evaluation of the SUMO Refinement Algorithm

Although the performance may be quantified in theoretical terms, these properties do not offer an insight into the practical issues of the algorithm. One factor not represented by these equations is the behaviour of the constraint solver when the set of relations R is incomplete, leading to the constraint solver making "guesses". These equations also assume that the user supplies the minimum information required (particularly for the best-case). It is possible that a user may supply more relations than necessary, for example providing relations that can already be deduced through the transitive properties of existing relations.

To model these aspects to better estimate the performance of SUMO, they are incorporated into empirical evaluation. Remodularisations produced by Bunch are refined using the original modularisation from a software system as the source of domain knowledge, with the mount of input from the original modularisation to refine the result measures. The evaluation was carried out on 150 simulated case studies, seeded from five Java software systems 30 simulated case studies are generated from each.

The research questions this evaluation aims to answer are as follows:

RQ1 How much manual effort is required for the SUMO algorithm to fully refine a modularisation, starting from Bunch?

RQ2 What is the relationship between the effort invested and quality obtained?

RQ3 What factors determine the amount of effort required for the algorithm to converge?

5.4.1 Methodology

In order to evaluate the effort required to use a refinement process, domain knowledge is required. To run a refinement process automatically, the domain knowledge must be available a-priori. This is made possible using existing expert decompositions from existing software systems.

The process is outlined in Figure 5.2. Authoritative decompositions are first extracted from each case study, which are then used to generate 30 "simulated authoritative decompositions". These decompositions are then used as the source of domain knowledge

Table 5.1: Case studies used in the experiments

Name	Description	Version	Size (SLoC)	Component Used	MDG Size (Nodes)	Number of Clusters
Collections	Apache Commons Collections library, consisting of classes that implement various data structures	3.2.1	151,727	`commons-collections-3.2.1.jar`	245	12
epubcheck	Java tool that validates ebooks in the `epub` file format.	3.0-b4	17,477	`epubcheck-3.0b4.jar`	75	13
JDOM	Java XML parsing library	1.0	10,972	`jdom-1.0.jar`	57	7
Wiquery	Library to integrate the jQuery JavaScript library with the Wicket Java web framework	1.5-M1	79,267	`wiquery-1.5-M1.jar`	204	37
ZXing	Java barcode scanning library.	2.1-SNAPSHOT	161,063	`core.jar`	184	29

that is given to the refinement algorithm which refines a modularisation produced by a clustering algorithm (this evaluation uses Bunch [81], although other algorithms may be used). These steps are detailed in the following sections.

5.4.2 Case Studies

Table 5.1 summarises the case studies. Although the size of the MDG is given, the simulated authoritative decomposition process described in Section 5.4.3 results in MDGs of varying sizes.

This evaluation focuses on the use of software dependency graphs, all of which were extracted from open source Java systems.

Requisite data must first be extracted from each case study: the dependency graph and its corresponding authoritative decomposition. The dependency graph is obtained with Dependency Finder [109]. The output must then be transformed to a graph format compatible with the modularisation algorithm that will be used to generate the starting points (discussed later in Section 5.4.4).

The existing package hierarchy of the classes in the system represents the authoritative decomposition, which is used as the target decomposition the SUMO algorithm must reproduce using the domain knowledge contained within it. To measure the upper bound of the effort required, the highest level of detail is selected such that the number of modules m is maximised, i.e. the bottom level of the hierarchy is used where packages are smallest. This represents the scenario where the user is aiming for the highest fidelity in the refined output, which will require the largest amount of information to produce. For these case studies, which are all Java systems, the package assignments can be obtained by splitting the fully qualified class name and taking the penultimate element. For example, from the fully qualified class name `org.jdom.output.DOMOutputter`, the `DOMOutputter` class will be placed in the `output` module,

This step then produces two artefacts: a dependency graph, which will undergo clustering and the lowest level package hierarchy, which will be used as the source of domain knowledge.

5.4.3 Simulated Authoritative Decompositions

A number of simulated software systems are generated from each case study, using the LimSim methodology [104]. This aims to avoid problems with a small number of case studies biasing the results of the experiment.

In the LimSim approach, random module dependency graphs and corresponding simulated expert systems are generated using the JRET [1] reverse engineering library. JRET randomly modifies the decomposition and makes corresponding changes to the dependency graph such that is similar but not identical to the original case study. This results in 30 simulated case studies, consisting of a dependency graph and authoritative decomposition.

5.4.4 Modularisation

The simulated systems are then clustered using the Bunch search-based clustering tool [81]. Although Bunch is used in these experiments, other algorithms could be used. Bunch was selected based on its versatility and its inclusion an API for programmatically running clustering operations. In addition, it has been widely studied in the literature including in an industrial context [40]. As Bunch is a stochastic algorithm, it is run more than once on each of the simulated case studies to avoid the bias introduced by selecting only one modularisation. Repeating the Bunch clustering process also allows for comparison of the performance of the refinement algorithm for varying qualities of the modularisation generated by Bunch.

A set of modularisations is produced by Bunch, one for each of the 30 repeated runs for each case study. As there are 30 simulated case studies for each real case study, this step results in a total of 900 modularisations, 30 for each of the 30 simulated module dependency graphs.

5.4.5 Simulating Interactive Refinement

This process uses the modularisation produced by Bunch in the previous step as the starting point for refinement. The simulated authoritative decomposition is used as the source of domain knowledge. The refinement process is detailed in Algorithm 7 and elaborated below.

Elements of the modularisation that are correct, i.e. the same in the generated modularisation and the expert modularisation are first locked. The sets of all relations in the Bunch result and the authoritative decompositions are computed and the intersection of these sets is then supplied to the constraint solver before the refinement phase begins. This models a scenario where the user has first indicated which components of the modularisation are correct before the refinement phase starts.

Once the refinement algorithm has been initialised with the correct components, the refinement process starts. The difference between the hypothesis and the expert modularisation is computed, then a subset of relations is supplied to the refinement algorithm,

Algorithm 7 Determine number of iterations required to refine a modularisation

```
expertRels ← getRelations(expertMod)
startingRels ← getRelations(startingMod)
correctRels ← expertRels ∩ startingRels
                    ▷ Apply all correct domain knowledge before measurement starts:
for rel ∈ correctRels do
    SUMO.addRelation(rel)
end for
                                                              ▷ Start refining:
count ← 0
MoJoLog ← []
repeat
    hypothesisMod ← getHypothesis()
    MoJoLog[count] ← MoJoDistance(hypothesisMod, expertMod)
    corrections ← (expertRels ∩ getRelations(hypothesisMod))^C
    numCorrections ← Poission(λ = 5)
    for i = 1 → numCorrections do
        addRelation(randomElement(corrections))
    end for
    count ← count + 1
until hypothesisMod = expertMod
return count
```

determined by a Poisson distribution with a mean of 5. The Poisson distribution is selected as it is anticipated that around 5 corrections would be supplied on average, with the long tail accounting for scenarios where, for example, the user identifies a whole cluster as being correct (discussed in Section 5.2.2).

In addition, the MoJoFM [124] distance between the hypothesis modularisation and the expert modularisation is computed and logged.

The process terminates when the hypothesis and expert modularisations are the same — i.e. when the refinement process is complete. The number of iterations taken is recorded and used to estimate the cost of the refinement.

5.4.6 Results

RQ1: Effort to Converge

The first research question assesses performance of the SUMO algorithm in terms of the required effort to totally refine the result. This effort is measured in terms of the number of iterations of the algorithm, and the number of corrective relations supplied.

Table 5.2 summarises the results, including the sizes of the simulated case studies and the number of iterations taken to produce a 25, 50, 75 and 100 percent improvement in

the modularisation, calculated by the change in MoJo distance between the hypothesis solution and the expert decomposition.

Table 5.2: Sizes of the mutant MDGs generated from each study and the number of steps required to reach each percentile of closeness to the target MDG

Case Study	Mutant MDG Size:				Reference modularisation:	Start MoJo	Iterations to Percentile: 25%		50%		75%		100%	
	Min	Max	Mean	St.Dev	Nº Modules	Mean	Mean	St.Dev	Mean	St.Dev	Mean	St.Dev	Mean	St.Dev
collections	453	523	492.57	14.96	12	73.38	14.56	2.47	21.99	3.58	31.47	4.46	43.23	6.29
epubcheck	138	191	159.13	12.92	13	35.43	8.05	2.03	13.58	3.22	21.50	5.26	31.30	6.57
jdom	124	183	146.57	12.68	7	9.91	4.02	1.72	6.24	2.18	8.31	2.90	9.67	3.24
wiquery	314	401	365.63	17.16	37	60.69	17.73	4.02	32.36	5.66	50.91	9.89	111.88	13.12
zxing	262	327	294.30	18.04	29	44.92	18.83	5.20	29.67	6.39	41.96	7.94	89.99	12.84

Total refinement of the modularisation requires a varying number of relations, as shown in the "Iterations to 100%" improvement column of Table 5.2. Figure 5.3 portrays this graphically in a violin plot. The figure shows a clear difference in the number of iterations to convergence between each case study.

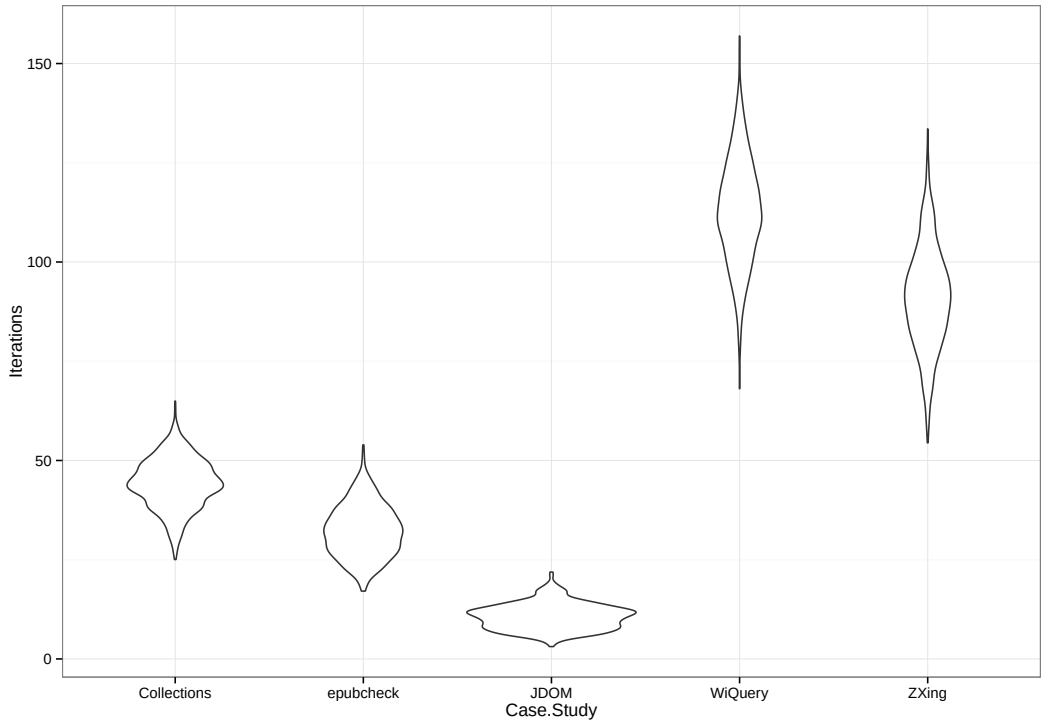


Figure 5.3: Violin plot of iterations required for convergence for each case study

Performance covers a wide range across all the case studies, with WiQuery requiring the greatest mean number of iterations of 111.88. In contrast, JDOM reached total refinement after fewer iterations, with a man of 9.67. The findings for this research

question show a large spread in the number of iterations of SUMO required to refine each of the case studies.

RQ2: Relationship Between Improvement and Effort

Although the number of iterations required to produce a complete refinement using SUMO is high in some cases, the profile of the improvement at each iteration offers an insight into the quality of the solution as SUMO improves it. Figure 5.4 shows a sample of 10 repetitions of the SUMO algorithm for each case study. It plots the improvement, in terms of the MoJo value over the iteration number. MoJo values are normalised to percentages, given by $percentage = \frac{max - value}{max}$.

The graphs in Figure 5.4 show improvement is experienced in the result almost immediately after domain knowledge is supplied to the algorithm. The majority show sigmoid curves to a varying degree — there is a point after which the improvement diminishes as extra information is added. This is particularly prominent in the ZXing and WiQuery case studies: around 80% of the improvement is observed after 50 iterations, the remaining 20% taking the same amount of time.

In each of the graphs, the improvement is observed to decrease at some steps. This may arise when the constraint solver selects a different solution between iterations. In cases where new constraints can already be inferred from those already provided, the same set of solutions will be available, of which the solver will select the most general. If there are more than one of these solutions, it is not guaranteed that the same will be returned each time the solver runs.

As the clusters are refined piece-wise using random constraints, the sequence of modifications may lead to solutions where the MoJo distance between the authoritative decomposition is higher. A correction may result in a large cluster being broken apart, for example. The improvement is not likely to occur over one or two iterations, as the corrections supplied appear randomly. Thus, a correction may reduce the MoJo score until other corrections related to it are provided later in the refinement process.

The outcome of this research question is that improvement in SUMO appears to follow a sigmoid curve, where the improvement depends on both the effort and the progress towards the result — improvement "tapers off" as convergence is approached.

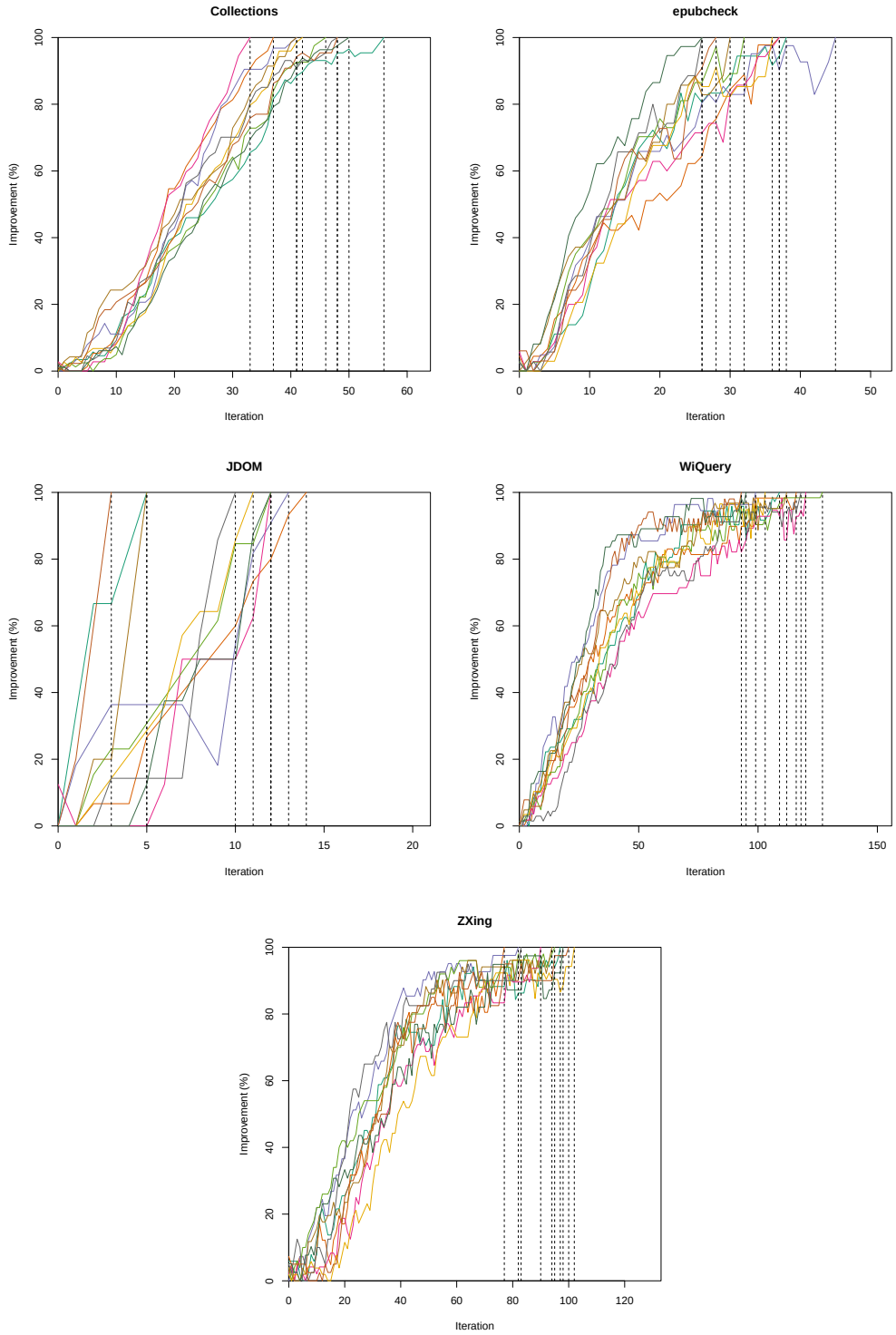


Figure 5.4: Improvement (percent) for 10 SUMO runs for each case study

RQ3: Factors Affecting Convergence Time

Figure 5.5 shows the quality of the modularisation versus the number of iterations taken to converge. As discussed in Section 5.4.6, the convergence time varies significantly among case studies. There is a correlation between initial MoJo quality and the number of iterations, which is less significant than the main determining factors: number of modules and MDG size.

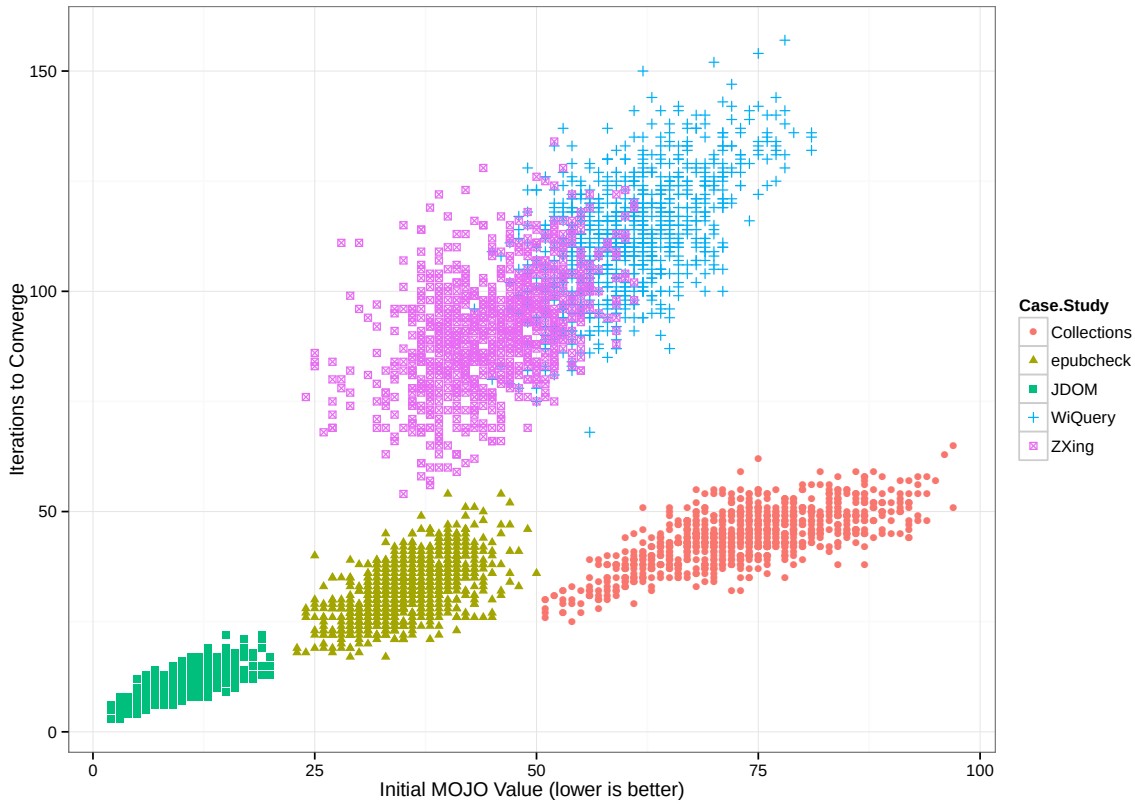


Figure 5.5: Scatter plot of iterations taken to converge versus MoJoFM score

The number of modules in the target has the highest correlation ($r = 0.958$). This is due to the amount of information required to separate nodes from one module, and is corroborated by Equation 5.7. Larger numbers of modules therefore require larger number of relations; in this case this causes the largest impact in terms of effort required. The time taken to converge is therefore dependent on the number of modules in the expert modularisation, the number of nodes and quality of the Bunch result, in that order.

Figure 5.6 shows the number of relations required plotted against the minimum number of relations for total refinement predicted using the equation. Results below the line, given by $y = x$ are cases where the number of relations taken is lower than the minimum number necessary to unambiguously reproduce the target modularisation.

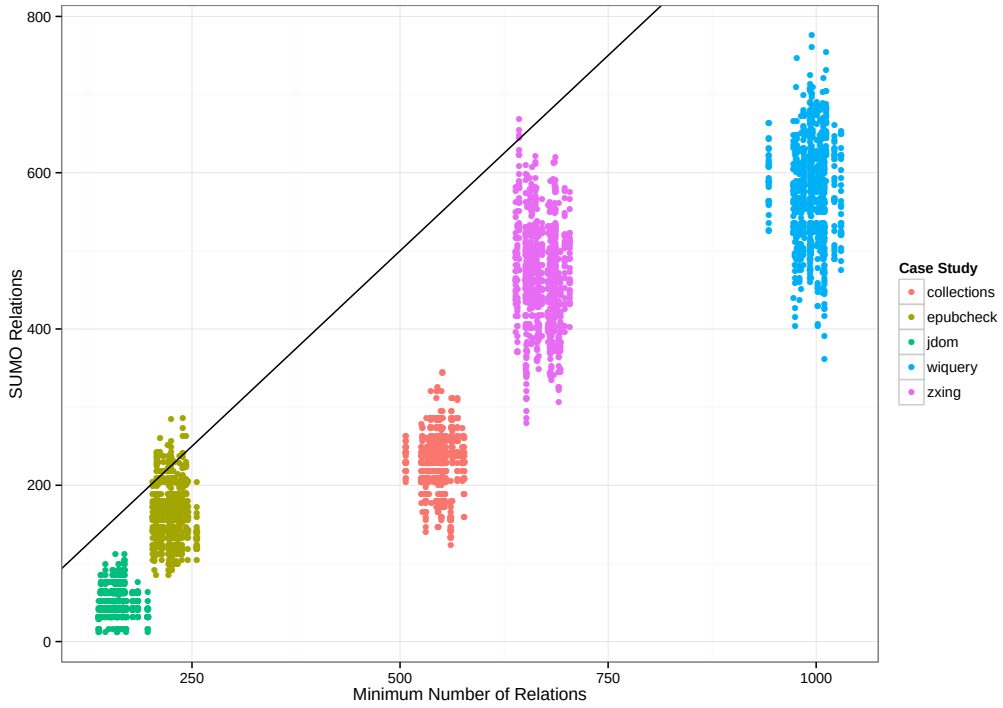


Figure 5.6: Scatter plot of relations taken to converge against minimum number of required relations, given by Equation 5.7

These results are attributable to two factors: the seeding stage of the evaluation (and therefore the quality of the Bunch input), and the constraint solver making a non-deterministic choice of the correct result from several solutions. The former is a more likely explanation of this phenomenon. In cases where MoJo values were lower (better), fewer corrections are required, as the correct relations are supplied prior to refinement, as described in Section 5.4.5. It is, however, not possible to distinguish between these two possibilities in the results.

In several instances for two case studies, epubcheck and zxing, SUMO exceeded the value estimated by Equation 5.7. These cases are evidence that more relations than required were supplied before SUMO converged. This can arise where relations are supplied that were already deducible; as the equation assumes no such deducibility, the random model may sometimes violate this assumption.

The conclusion for this research question is that convergence time depends on a number of factors, with the number of modules having the greatest affect, followed by case study size and finally the quality of the Bunch result.

5.4.7 Discussion

Although the evaluation shows that SUMO requires a large investment in terms of the number of iterations required to reach convergence, the results for the second research question show that improvement is gained at regular intervals throughout the refinement process. As SUMO is an interactive algorithm, it is possible for it to be terminated at any point, rather than being run to completion. The profile of the improvement shown in Figure 5.4 suggests that a large portion of the improvement occurs earlier in the refinement process. Combined with the interactive approach adopted in SUMO, this incremental improvement characteristic makes it highly applicable in scenarios where an observable improvement is required but the availability of domain expert time is limited.

The amount of information was determined to depend most on the number of modules in the input, followed by the number of modules. Thus, a more detailed result requires more relations to produce it. This is due to the transitivity of a positive relation; relations added for a member of a group of positively linked elements. Subsequent relations added to one element of such a large group apply to each element, allowing large portions of the search space to be ruled out with a smaller incremental investment.

5.4.8 Threats to Validity

There are several factors which constitute threats to validity for these experiments:

The constraint solver may impart bias. There are many possible solutions which conform to the set of constraints, particularly at the start of the process when there are few constraints. The solution selected by the solver impacts the direction of the search, as the subsequent set of constraints depends on the hypothesis solution. The constraint solver's strategy for selection from this set makes the results dependent on it; it is therefore not possible to state that this performance would be typical of all constraint solvers used.

The use of Bunch-generated starting points could bias the results. As Bunch is a stochastic algorithm, the quality of its output may non-deterministically vary. The risk of this causing bias is mitigated by using 30 Bunch-produced modularisations for each MDG.

The simulated responses may not be characteristic of a human user. The number of constraints, modelled with a Poisson distribution may in practice not match the performance of a human user. Moreover, the types of constraints, and the specific constraints (made through random choice in these experiments) may not be consistent with the choices made by a human. A human user is likely to use their intuition to make better choices, so this evaluation may not suitably estimate the actual performance of the tool in a realistic use case.

The subject systems, synthesised MDGs and decompositions may not be representative of software systems. The evaluation was conducted on five software systems, the majority of which were libraries. It is not possible to extrapolate from such a small sample to make general assertions about the performance of SUMO. The use of the LimSim's mutation approach is used to generate a larger population of case studies, however the random mutations applied may risk removing elements of the original systems which may make them harder to modularise.

These threats to validity do not permit conclusions to be drawn about the general performance, the findings for each of the research questions, exploring facets of SUMO's performance, can be justified however, through the diverse case studies used and the further diversification gained through the use of the LimSim methodology.

§ 5.5 Conclusions

This chapter introduced the SUMO constraint-based remodularisation algorithm that allows the gap between automatically generated modularisations that maximise metrics and human intuition to be bridged using an interactive approach. The performance of the approach was quantified in theoretical terms as well as more practical terms using simulated authoritative decompositions, showing that although the cost of total refinement is high, refinement can be obtained within several iterations of the algorithm.

The evaluation of the SUMO algorithm has demonstrated a practical means to improve upon the clusterings produced by other remodularisation tools. The refinement approach was demonstrated to work on top of results produced by Bunch, but the flexibility of the use of SUMO as a refinement step makes it applicable to other remodularisation approaches, including Crunch, or other hierarchical clustering algorithms.

Alternative methods of supplying constraints may be beneficial to allow scenarios where a large amount of positive information has been added, as discussed in Section 5.4.7. By reducing the cost of supplying such constraints, such as the use of a lock command, the effort required may be further reduced. An implementation of such a measure is demonstrated in the GUI version of the SUMO tool, which is evaluated in a user study in the following chapter.

General inferences about the performance of SUMO cannot be made from the study in this chapter, however. The random constraints supplied in the evaluation may not model a typical user. Despite this, the evaluation has demonstrated that SUMO potentially offers a viable means to improve upon an existing modularisation. The following chapter expands upon this study by quantifying the practical performance of SUMO in a realistic scenario with human users.

Chapter 6

Human Factors Affecting Supervised Remodularisation

The previous chapter described and evaluated the SUMO algorithm, showing how domain knowledge could be exploited to produce refined modularisations to more closely match an expert's intuitions. Evaluation of the approach determined that the improvement obtained is incremental, although not monotonic, using a simplistic model of the feedback given by a user. Convergence was shown to require a large number of iterations, although this incremental performance showed that SUMO may be applicable in scenarios where running it to completion would be unfeasible due to the volume of required domain knowledge.

This chapter details enhancements to the SUMO algorithm that have been made to adapt it for use in an interactive graphical tool that allows a user to interactively refine a modularisation. A pilot study, conducted at the University of Sheffield, is described which identified these enhancements and tested the methodology described in this chapter. This methodology, together with the revised tool are then applied in an empirical evaluation. This study was conducted at the University of Leicester and further investigates the performance of SUMO for the remodularisation of a component from the open source Google Guava library, determining that all of the 35 participants were able to use it to refine a Bunch-generated modularisation within 80 minutes, even without prior experience with the case study.

This chapter makes the following contributions:

1 A graphical implementation of the SUMO algorithm and the addition of a constraint solver "seeding" strategy to improve the quality of the intermediate results it produces

2 A user study, conducted with 35 participants that replicates a typical remodularisation procedure, that determines improvement was achieved for all participants for a 122 class remodularisation task with a mean completion time of one hour.

This chapter is structured around a study that aims to validate the findings of the empirical evaluation conducted in the previous chapter when the SUMO tool is used by human users. The research aims of the study are first described in the form of research questions, before the revisions to the SUMO tool and algorithm are described. The methodology applied in both the pilot study and the empirical study is then described, highlighting the changes made as a result of experienced gained in the pilot study. Finally, the results are presented and their implications discussed with respect to the three research questions.

§ 6.1 Research Questions

The objective of this study is to quantify the performance of the SUMO algorithm in a realistic setting, using real users. The research questions remain the same as those for the previous study conducted in Chapter 5. Although the research questions are the same, necessary changes to the methodology as a result of the use of human users changes the data available to address them. These changes are described in Section 6.3.

The research questions addressed by this study are as follows:

RQ1 How much manual effort is required for the SUMO algorithm to produce a refined modularisation, starting from Bunch?

RQ2 What is the relationship between the effort invested and quality obtained?

RQ3 What factors most affect the convergence time?

§ 6.2 The SUMO Tool

Figure 6.1 shows the interface developed for the SUMO tool. Modules are represented by large coloured polygons which encapsulate labelled boxes which represent elements (classes or states, for example). The user adds domain knowledge through the creation of edges between modules, which are shown in red or green, which represent negative and positive relations respectively. New modularisations are only computed when the user requests them, via the "Next" button in order to allow the user to construct a hypothesis, or modify or delete edges, before the configuration of the modules is recomputed.

The graph layout is built on the Prefuse [49] visualisation framework and dynamically adjusts to the configuration of the relations. Prefuse was selected as it provides a flexible framework for visualisation beyond the graph representation used in this version of the SUMO tool. Each edge type carries a different weight, which causes the arrangement to adjust as the user supplies more information. Positive edges have the strongest weight, causing elements to be drawn together when the user adds the relation.

Documentation can be viewed during a remodularisation task in the lower right panel. Hovering the mouse cursor over one of the elements in the main visualisation panel

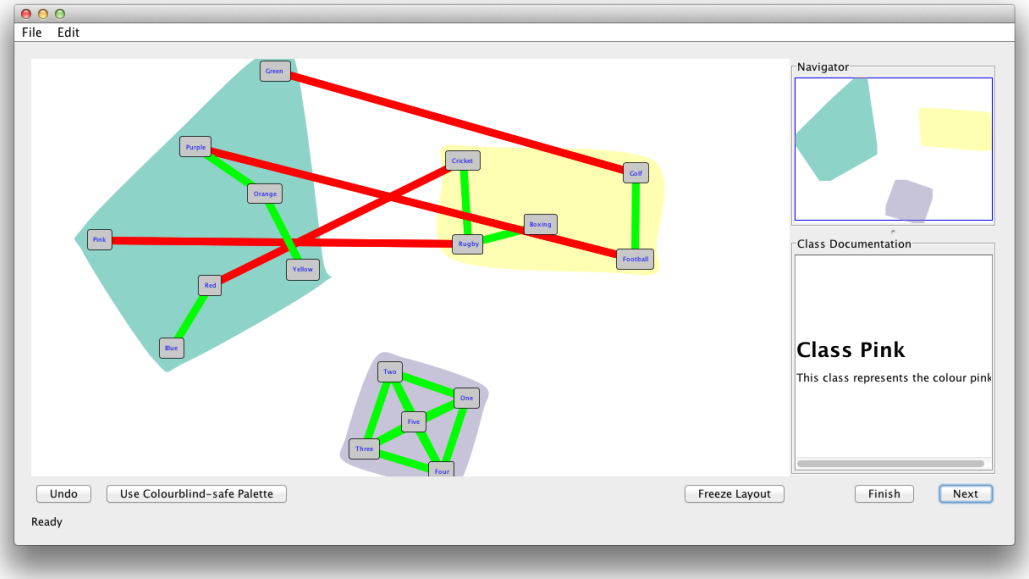


Figure 6.1: The SUMO user interface

causes its documentation to load in this area. This allows the documentation to aid the user to make decisions during refinement, enabling reverse engineering of the system to be integrated into the process. The panel used can show any HTML documentation that can be generated for the system. In these experiments, the Javadoc class documentation is used.

Locking

SUMO allows a cluster to be locked to prevent further modification by the constraint solver. This lock command can be used by the user to indicate that a module is correct and cannot be further refined. The lock operation automatically inserts positive relations between all pairs of classes in the module, and negative information between each class within and each class outside the module. This enables the user a large number of candidate solutions to be ruled out with a very low investment of effort, versus the manual addition of all the constraints automatically applied.

Undo

In addition to the lock command, the tool also allows the user to return to a previous modularisation, abandoning any changes since it was presented. This allows backtracking when mistakes have been made and can also be used to resolve conflicts instead of locating and deleting the erroneous relation. The current SUMO implementation does not indicate which relation causes the conflict, requiring the user to manually identify the fault.

6.2.1 Seeding the Constraint Solver

A common issue identified by participants of the pilot study (detailed in Section 6.3.5) was the poor quality of the modularisations produced by the solver when an improved modularisation was requested (via the "Next" button in the interface). This arose where the solver returned an overly general result when the user had not provided any counter-examples. For example, adding a large number of positive edges and clicking the next button produced a solution containing one monolithic cluster as no counter-examples were provided. Although this solution matched the constraints, it was counter-intuitive.

In order to remove this source of confusion, the constraint solver can be seeded with the previous modularisation. The previously described version of the SUMO algorithm leaves assignment of classes to modules entirely to the constraint solver which results in this misbehaviour. Pre-seeding the state of the solver causes it to search from the previous solution, rather than a single-cluster solution, resulting in new solutions that are closer to old solutions that still conform to the set of supplied constraints.

Algorithm 8 Constraint solver initialisation and use

```
global attempts ← {}, seeds ← {}
global MAX_ATTEMPTS ← 100
function getHypothesis(relations, previousModularisation)
                                                        ▷ Generate seeds:
    for module ∈ previousModularisation do
        num ← random()
        for class ∈ module do
            seeds[class] ← num
        end for
    end for
    add all relations to the solver
    return solver.solve()
    if problem is not solvable then
        return error
    end if
end function
function getInitialAssignment(class)
    if attempts[class] < MAX_ATTEMPTS then
     attempts[class] ← attempts[class] + 1
        return seeds[class]
    else                          ▷ Stop after the limit is exceeded for this variable
        return random()
    end if
end function
```

The seeding process consists of modifying the initial configuration of the problem before the constraint solver runs. Each variable, which assigns the module to the class as described in Section 5.2.3, is assigned from a pool of numbers mapping to modules from the previous solution. For example, if the two classes a and b were in the same module, the starting configuration of the solver would be $a, b = N$, where N is a randomly selected number which represents the module. This is implemented as a modified value selector for the CHOCO [55] constraint solver and is described in Algorithm 8. The implementation of this seeding strategy makes a small adjustment to the $getHypothesis$ function (described previously in Section 5.2.2).

This pre-seeding is restricted to 100 assignments for each class; after this limit is exceeded, a random result is returned instead. This limit represents a trade-off between the quality of the solution produced and the response time between the user requesting an updated modularisation and the SUMO algorithm returning one.

6.2.2 Committed Edge Visualisation

The implementation only allows edges that are not incorporated into the hypothesis solution to be modified, such that using the "Next" operation locks the relations, unless the undo button is pressed. In the pilot study, there was no distinction made between these "committed" edges and mutable edges. Without a means to see which edges were still mutable, two problem scenarios arose.

The first of these problems was that the information discarded by the undo operation was not visualised before it was carried out. This made predicting the behaviour of the undo command difficult for the users in the pilot study. Secondly, in cases where conflicts arose, it was difficult to determine which edges to start checking for conflicts. These two factors made conflict resolution a highly costly activity.

In response to this, conflict resolution was made easier by reflecting the mutability of edges in the colour saturation of the edges. Edges that cannot be modified are faded out, while the uncommitted edges remain the normal red or green colours. This enables the current "working set" of edges to be seen much more clearly.

6.2.3 The Revised SUMO Algorithm

Algorithm 9 summarises the process of refinement using the SUMO tool and shows how the revised $getHypothesis$ functionality integrates with the GUI, as well as how the undo and committed edge visualisation is implemented.

Algorithm 9 Implementation of the SUMO Tool

```
function SUMO(previousModularisation)
    while last action was not finish do
        if last action was undo then
            restore previousModularisation and relations from history
        else if last action was next then
                                        ▷ Translate UI relations to pairwise constraints:
            relations ← fetchRelations()
            newModularisation ← getHypothesis(relations, previousModularisation)
            if newModularisation = error then
                show error dialog
            else
                previousModularisation ← newModularisation
                show previousModularisation
                for relation ∈ relations do
                                        ▷ Show committed edges in a different shade:
                    relation.fadeOut()
                end for
            end if
        end if
    end while
end function
```

§ 6.3 Methodology

The experiment consists of two distinct stages: instructor-led training, followed by interaction with the tool to complete two modularisation tasks; one tutorial and then the case study of the experiment. The latter stage was conducted under exam conditions with each participant using identical hardware running a locked-down environment which only provided access to the SUMO tool. These measures were taken to ensure ensure the experience for each user was as similar as possible.

This section firstly describes the prototype used for both the pilot study and the main experiment; the details and outcomes of each study are discussed in the subsequent sections.

The experiment followed the following sequence:

Stage 1: Tutorial (10 minutes) A non-interactive presentation providing an overview of the tool and its features

Stage 2: Live Demonstration (10 minutes) A session of interaction with the tool was shown in real-time with commentary

Stage 3: Example Task (approx. 15 minutes) Participants performed a remodularisation task on an example MDG

Stage 4: Main Task (up to 80 minutes) Participants used SUMO to remodularise the case study

Each step of the experiment is detailed in the following sections.

6.3.1 Stage 1: Tutorial

The non-interactive tutorial consisted of a 10 minute presentation with slides describing the motivations of the experiment. This included a short description of the remodularisation problem, followed by an overview of the features implemented in the tool. The participants were also instructed that the button to end the task should be used when they were "happy with the package structure". This wording was selected to avoid priming the participants. Tutorial sheets were circulated that provided an overview of the function of each component of the tool—it is attached in Section A.

6.3.2 Stage 2: Live Demonstration

Following the introductory presentation, the participants were shown the SUMO tool running live. An instructor interacted with the tool, vocalising the process and elaborating the functionality used in the tool. This demonstration included the use of the tool to remodularise parts of a toy problem in order to demonstrate the various features available. The example consisted of elements drawn from three distinct categories: company names, place names, and forenames:

Module 1	Tom, Phil, Robert
Module 2	Lisa, Harry, Paul
Module 3	Jennifer, Maria, David
Module 4	IBM, Intel, Microsoft, Anna
Module 5	Apple, Sheffield, Leicester
Module 6	London, Google

The example was kept as abstract from software as possible to avoid predisposing the participants to using a particular judgement of similarity beyond their own intuition.

6.3.3 Stage 3: Example Task

Following a short break, the participants started using the SUMO tool on a short example task. This task was designed to familiarise them with the tool and as such the MDG was easy to modularise by design. The criterion for similarity was not described to the participants, in order to avoid predisposing them to one solution or another. Exam conditions were instated before this task began. In cases where participants asked questions during the assessed task, answers relating to the problem were not given — only questions about the tool's behaviour or its functionality were answered.

The example task contained 17 "classes", each of which was taken from one of three distinct categories: sports, colours, and numbers. The initial assignments of classes to groups for the example task were as follows:

Module 1	Red, Blue Green
Module 2	One, Two, Three, Four, Five
Module 3	Yellow
Module 4	Football, Orange, Purple, Golf
Module 5	Rugby, Pink, Boxing
Module 6	Cricket

Module 1 was designed to encourage users to exercise the lock option, as it is already complete. The starting configuration of this example was designed to demonstrate that the modularisation may contain varying degrees of accuracy, and to emphasise that singleton modules (containing only one class) were possible.

6.3.4 Stage 4: Main Task

In the previous study, five case studies were used, from which 30 simulated systems were produced, each clustered 30 times with Bunch. This number of case studies cannot be assessed with a small pool of participants, thus requiring a considerably smaller pool of case studies. Environmental factors must be controlled to the extent possible;

The participants then performed the main task in the same way as the previous example task. The case study used differed between the pilot study and the main experiment. This part of the experiment was conducted under exam conditions. In cases where participants asked questions, only those asking for clarification about the behaviour of the tool were answered; queries specific to the case study were not answered.

The following subsection describes a pilot study that was conducted at the Department of Computer Science at the University of Sheffield that was used to test the methodology used in the main experiment.

6.3.5 Pilot Study

This pilot study followed the methodology described in the previous subsections the same process, minus the live demonstration phase and the help-sheet. The pilot study was also conducted on a smaller case study, JDOM, which is described in Section 6.3.5. The version of the tool used for the pilot study used a previous implementation of the SUMO algorithm that did not use the seeding strategy or committed edge visualisation described in Section 6.2.

The pilot study was conducted with 22 participants, 8 of which were from industry. Of the other participants, 4 were undergraduates, 7 were PhD students, two were research associates and one was a lecturer. The non-industrial participants were all recruited from the Department of Computer Science at the University of Sheffield.

Tutorial

The tutorial phase of the pilot study omitted the live demonstration of the use of the tool, moving straight to the example task from the initial presentation. This resulted in a high number of questions during this step, prompting the addition of a live demonstration to the main study, as well as an information sheet.

Case Study

For the pilot study, version 1.0 of the JDOM XML parsing library was used. Only classes in the `org.jdom` package and below were used, excluding inner classes. After filtering on the namespace and inner classes, the case study consisted of 57 classes. Only class names were made available to the participants of the study to remove cues that may have primed the participants.

The documentation was extracted from JDOM's Javadoc class documentation, which was sanitised by replacing all qualified references to classes (those including package names) with only the class name.

Rather than using a particular clustering algorithm as a starting point, the participants were given the same random modularisation to refine. This avoided the bias introduced by the clustering algorithm — as seen by the results in the previous chapter, the performance of the clustering algorithm has an impact on the number of relations. The same starting modularisation was provided to each of the participants, generated using the JRET library and consisted of five groups of classes.

Outcomes of the Pilot Study

The majority of participants appeared to avoid the use of the "Next" button which advanced to the next SUMO-generated solution. Several of the participants had queries about when to use the button, others opted to only provide positive information, resulting an unsatisfactory solution (where all elements were clustered into one large unit). In several cases, users chose to provide as many edges as possible between presses of the "Next" button.

One participant commented that they would prefer to build their own modules, rather than manipulating an existing modularisation using constraints. Another suggested that the starting configuration should be one monolithic module, which would be divided by the relations supplied by the user.

In several cases, users had trouble distinguishing between information that was already committed to the constraint solver versus added since the last run of the solver. This ultimately led to difficulty determining which relations needed to be removed to rectify a conflict of constraints.

The problems identified in the pilot study led to the addition of the live demonstration phase in the main empirical study, as well as the use of a printed handout which summarised the functionality of each of the elements of the SUMO interface. This

help sheet is attached in the appendix in Section A. The following section describes, and discusses the results of an empirical study that incorporates the changes made in response to this pilot study.

§ 6.4 Empirical Study

This experiment followed the methodology described in Section 6.3, including the use of a live demonstration stage. The experiment was conducted at the Department of Computer Science and Maths at the University of Leicester.

35 participants were recruited for the experiment, 4 of which were academics (Lecturer or Research Associates), two were PhD students, one was a fourth year undergraduate and the remainder were Masters students studying Software Engineering. One of the participants reported that they had "used Guava before". These participants are individuals with computer science backgrounds. All but one had not used the Guava library before and thus had no previous domain knowledge before starting the task.

Case Study

The Collections component of the Google Guava Java library [1] was selected for the case study. The Collections component of the library includes implementations for various data structures not present in the standard Java Collections Framework. It provides a comprehensive set of class documentation. The Collections component itself is a package in Guava (its classes are contained within `com.google.common.collect`). Version r-06 was used for the case study, and contains 122 classes.

The task for the empirical study was to remodularise the Guava Collections component. Similarly to the experiments run in Chapter 5, the Bunch [78] remodularisation algorithm was used to produce the starting modularisation to be refined by the participants. In contrast to the automated experiments from the previous chapter, the median level of hierarchy was used from the Bunch results, rather than the lowest level. This change was made to ensure the experiments assessed the practicality of the SUMO tool in a scenario as close to a real-world setting as possible. The median was selected on Mitchell's recommendation [78, p.99], which was justified by the balance between cluster size and the number of clusters produced.

6.4.1 Results

The results from the main task of the empirical study are outlined in the following sections, grouped by the relevant research question.

[1] `http://code.google.com/p/guava-libraries/`

RQ1: Effort Required for Remodularisation

Figure 6.2 shows a histogram of the amount of time taken to complete the main task, from viewing the first modularisation to when the final modularisation was presented. The majority of the participants completed the task within an hour; the mean time to complete the task was 54 minutes and 18 seconds, with a standard deviation of 704 seconds.

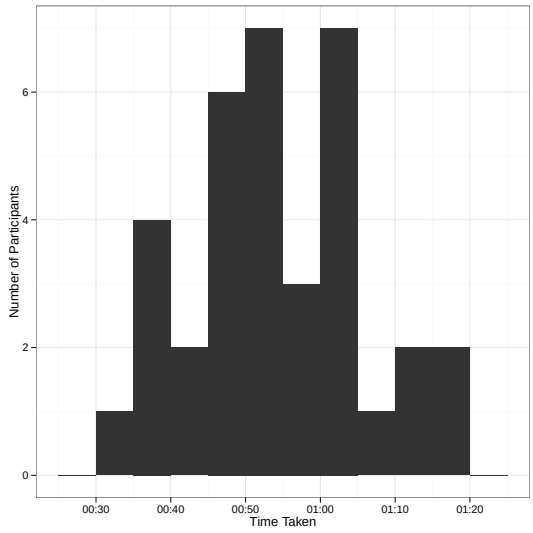


Figure 6.2: Histogram of time taken to finish the main task

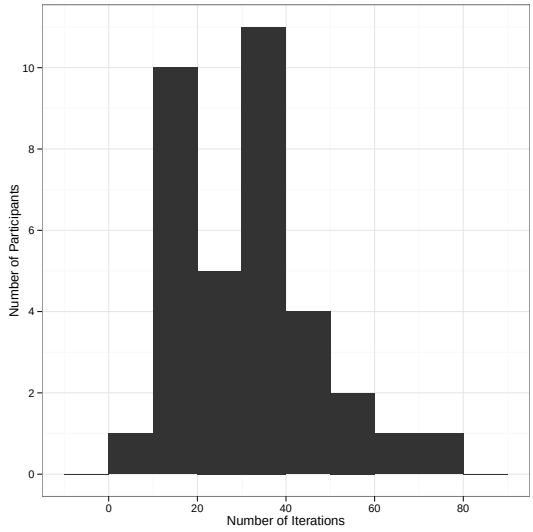


Figure 6.3: Histogram of iterations taken to finish the main task

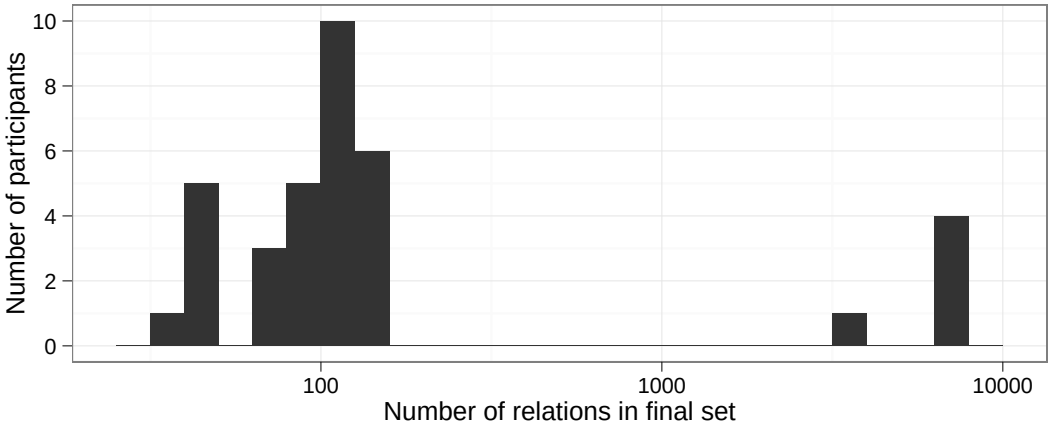


Figure 6.4: Number of relations provided by participants

The distribution of the number of iterations taken to reach convergence is different to the time taken. Shown in Figure 6.3, the median number of iterations was 30, mean 30.857 and standard deviation 16.714.

The information required by the SUMO algorithm to converge is shown in histograms in Figure 6.4. The majority of participants provided fewer than 150 relations, while several provided a considerably larger amount. The median number of relations provided by all users was 114, while the mean was 1037, skewed by the five participants who provided a larger number of relations, resulting in the large standard deviation of 2399. These five participants did not manually provide this high number of relations, however, they were added by the lock functionality of the tool.

A stacked bar chart of the sizes of the final relation sets for the users who did not use the lock function is given in Figure 6.5. Only the relations for the users who did not use the lock operation are shown in this figure, as all the relations provided by users who utilised the lock operation were synthetic and do not represent the manual effort invested. In this figure, the number of negative relations supplied is much lower than the number of positive relations for all participants. Of the 30 participants who did not use the lock operation, 17 provided at least one negative relation, the remainder only supplied positive information to correct the modularisation.

In the context of the research question, the finding is that the effort in terms of time, iterations and relations varied between the users significantly. Only a small subset (five) of the participants used the "lock" functionality of the tool.

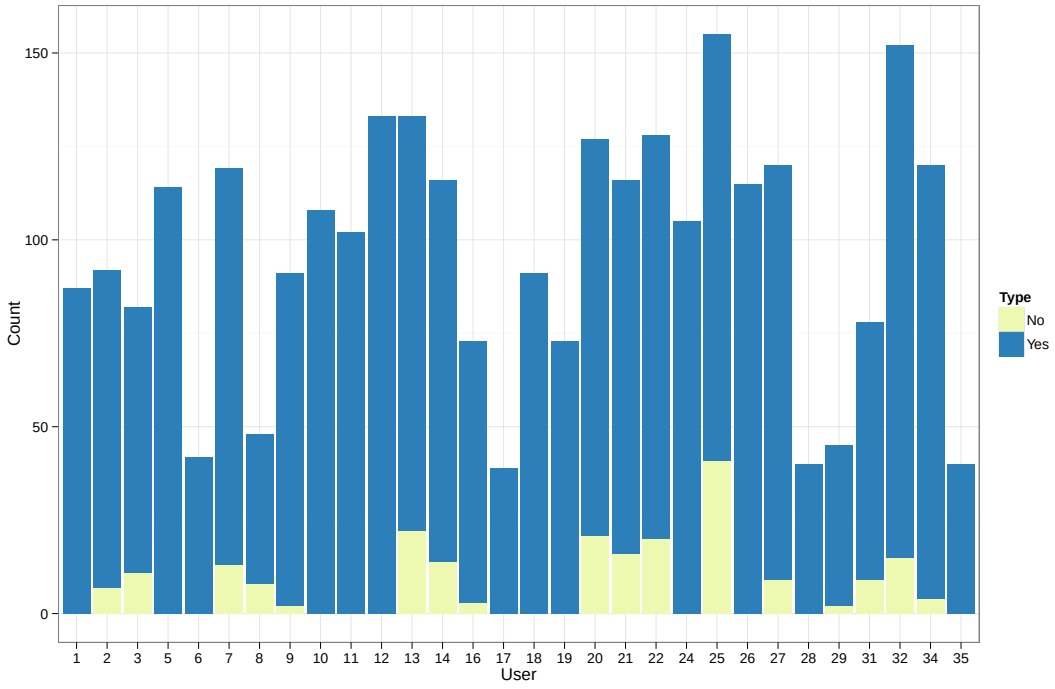


Figure 6.5: Stacked bar chart of number of relations provided by users

RQ2: Relationship Between Effort and Quality

Figure 6.6 shows a line chart of the Rand measure between the final modularisation and the modularisation at each point the user requested a new solution from SUMO.

The chart shows a large difference in the final modularisations produced by each user and the initial configuration. This firstly represents a disagreement between each user and Bunch, which serves as justification for the use of interactive refinement. Secondly, the disagreement with Bunch is different between users: in some cases the modularisation is highly similar while it is lower for others. This demonstrates the necessity for the refinement step and shows that the remodularisation problem is subjective.

The improvement, calculated by $\frac{max - value}{max}$ per iteration (shown graphically in Figures 6.6 and 6.7) shows a general trend of an overall increase in quality over time, however the trajectory followed differs greatly between individual users. This contrasts to the results from the experiments in Chapter 5, where a sigmoid curve was visible for most of the case studies. A pattern of low or no improvement for the first several minutes of the task was observed, however; this pattern was also observed in the previous study in Chapter 5. After this initial period, the improvements gained generally increase as time increases.

The conclusion for this research question is thus the same as that for this research question in the study in Chapter 5: intermediate improvement occurs regularly, however the improvement is less predictable and lacks the clear sigmoid shape that was previously observed.

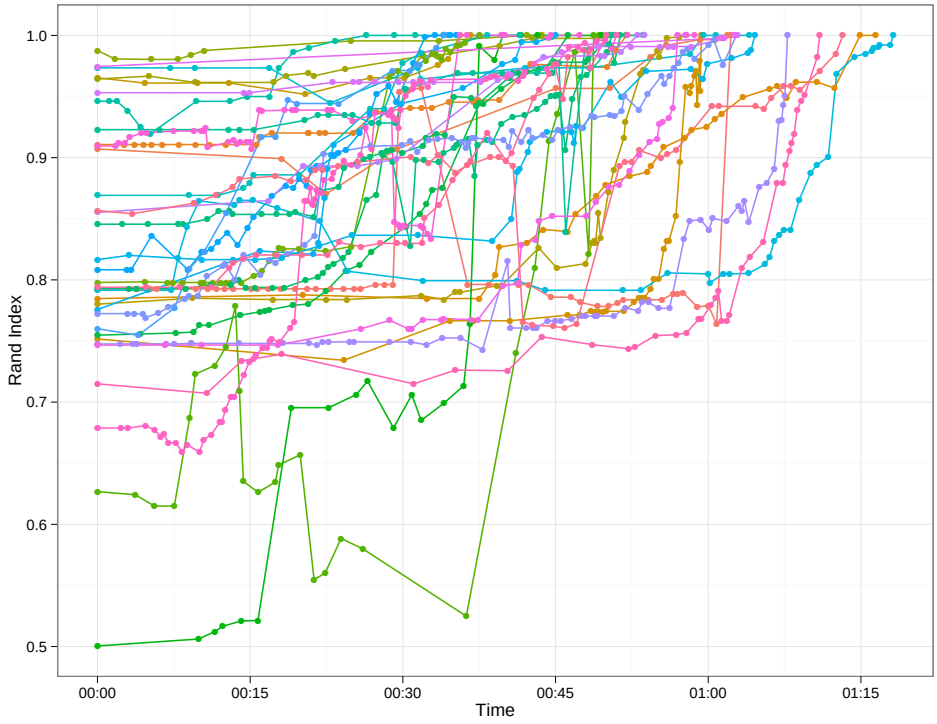


Figure 6.6: Rand similarity for each package and the final package over time

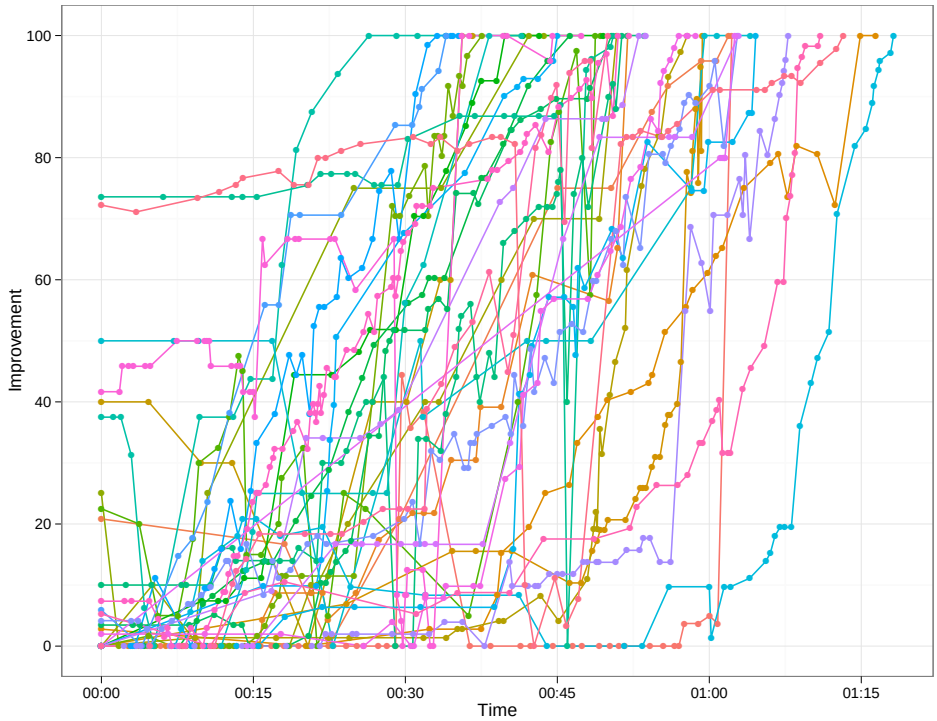


Figure 6.7: Improvement over time for each iteration

RQ3: Factors Affecting Convergence Time

The number of packages requested from SUMO before convergence is plotted against the time taken for each user in Figure 6.8. It shows little correlation between the number of packages requested and convergence time; some participants using considerably more iterations took the same time as other participants who used fewer iterations. This lack of correlation suggests that the number of iterations of the SUMO algorithm taken is not a good predictor of time taken (or vice-versa).

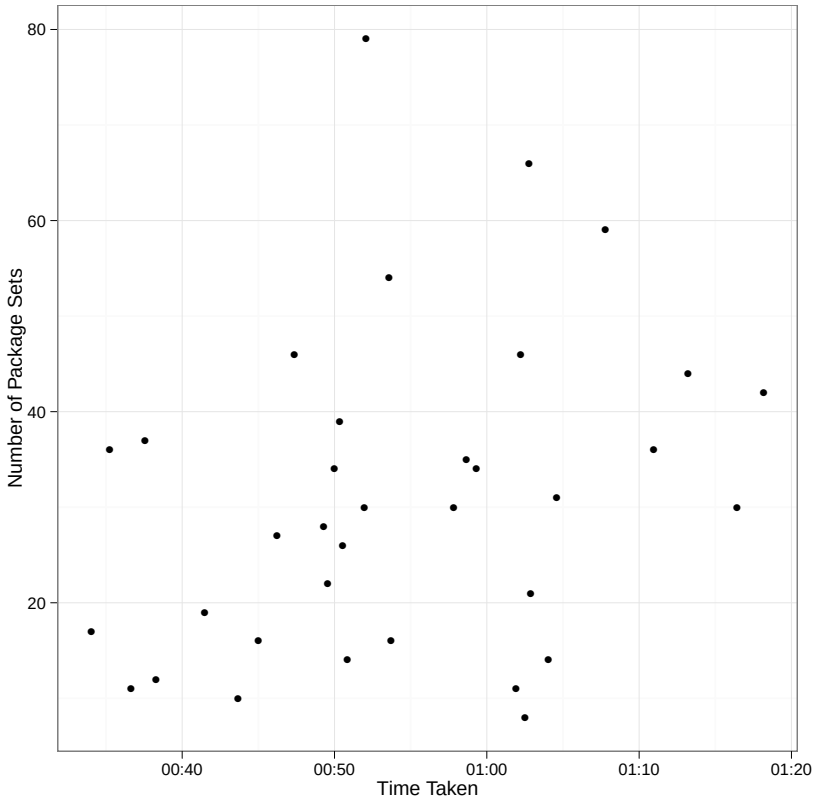


Figure 6.8: Scatter plot of time taken versus number of iterations

The time series data shown in Figures 6.6 and 6.7 show that the starting points, which were all identical, have different similarities for different users. This difference is evidence that the users each individually produced different solutions after using SUMO.

One of the participants indicated that they had used Google Guava before in the survey. This participant took 58 minutes and 40 seconds to complete the task, using 35 iterations of the SUMO process. This places the participant close to the average (both by median and mean) for both time taken and iterations.

The outcome for this research question is therefore less conclusive; the individual appears to be the most significant determinant of convergence time, which is corroborated by the evidence that each user was producing a different modularisation.

§ 6.5 Discussion

The experimental results show that the SUMO tool is capable of enabling 35 participants to remodularise an unfamiliar piece of software consisting of 122 classes within an hour for the majority of participants.

The profile of improvement did not exhibit the cohesive patterns observed in the previous study, however there is an observable trend at the start of the process of several minutes of little improvement, after which the majority of participants started to produce improved modularisations with the tool. This pattern was observed in the previous study to an extent, suggesting that there is a minimum number of relations required before the regular improvement is observed.

This initial period could be attributed to the participants familiarising themselves with the system. Beyond this point, the modularisations for each user begin to improve, with reductions in quality reversing quickly for most users. This improvement profile suggests that SUMO can be used to produce improvements in scenarios where a small investment of expert can be made that at least exceeds this initial low improvement period.

The number of negative relations supplied by each user varied, although the majority of the relations were positive, i.e. the users mostly identified elements where Bunch separated two elements that should have appeared together. High proportions of positive relationships represent the most "power", through their transitive property, which suggests that the constraint solving approach was able to exploit this property in a large number of cases. The use of the lock operation by the participants also suggests that automating the addition of constraints at the user's direction is a beneficial addition.

The data collected in this experiment shows strong evidence that the user is the strongest source of variance for the modularisation task. This can be explained in part by the subjectivity of the "correct" modularisation. Although each user is starting from the same point, the target varies between the individual, thus the paths taken to reach it will differ. This possibly explains why the profile of improvement is more varied than in the automated experiments in Chapter 5, in which the target modularisation was a highly similar decomposition (produced by LimSim from the package structure).

Overall, this study has empirically demonstrated of the efficacy of the SUMO approach and its implementation in the interactive tool. All participants were able to use it to produce an improved modularisation for a piece of software they were unfamiliar with. An important observation is that the participants were not domain experts—34 of the 35 users had no experience with the case study, but the SUMO tool still enabled them to tailor the existing modularisation to the domain knowledge they acquired from the system during the task. This suggests that the user of the SUMO tool may be able to complete a task by acquiring knowledge on an as-needed basis, which further reduces the potential cost and extends the applicability of the approach as component in a reverse engineering exercise.

6.5.1 Threats to Validity

As the study uses humans, there are a variety of threats to validity to consider:

User interface design may bias the results As the time taken to use the tool is dependent on knowing how to use it, a user's proficiency may have an impact on the amount of taken. The users all started from the same level of expertise with the tool (i.e. none) meaning there is no threat that a user may be more acquainted with the tool than another. It is not possible, however, to assert that the user interface is perfect — the results of the experiment are inextricably tied to the full "stack" of SUMO and the UI used and cannot be used to make a general assertion about the performance of human use of SUMO with any arbitrary user interface.

The individual user may bias the results This threat is mitigated by the number of participants and the wide demographic. To ensure the use of colour to represent relations in the visualisation did not impact users with impaired colour vision, the tool allowed the palette to be switched to ensure users would be equally able to use the tool to the best of their ability.

The final result may not be the user's desired solution The experiment assumed the user would only finish the task when they were satisfied with the result. However, fatigue or other factors may have resulted in them ending the task prematurely. The participants were invited to take a short break between the tutorial and the main task to minimise the risk of fatigue. As the previous experiments showed, SUMO will produce improvements in the modularisation provided the input information is correct. The time series plots in Figure 6.6 and 6.7 show that an overall improvement is experienced, thus, if the hypothesis that each participant provided information to the best of their ability (and did not elect to deliberately supply incorrect relations) holds, the algorithm produced a result which is superior to the original configuration in all cases, however the final configuration may not represent the global optimum that each user would produce with the tool.

§ 6.6 Conclusions

This chapter introduced a graphical implementation of the SUMO algorithm and presented an approach to improving its performance through the use of a seeding technique for the constraint solver. This modification was made in response to comments received during a pilot study which found that the unmodified version would produce unexpected solutions to the set of relations supplied by the user.

The empirical study demonstrates the necessity of the incorporation of domain knowledge into the modularisation process, evidenced by the different solutions produced by the users in the empirical study; each participant produced a different refinement of the original Bunch-produced modularisation. The practicality of the SUMO approach

is demonstrated through the use of a realistic case study, consisting of 122 classes. The majority of the participants completed the task in around an hour, having supplied fewer than 150 relations to produce a result tailored to their intuitions. This, coupled with the partial improvement observed in the interim steps, demonstrates SUMO's capability to enrich automated remodularisation techniques, even in scenarios where running the algorithm to convergence is unfeasible.

The following chapter concludes the thesis, showing how the work in this chapter integrates with the work in Chapters 3 and 4 to represent an enhancement to existing modularisation approaches and outlining future avenues of exploration that extend from it.

Chapter 7

Conclusions & Future Work

§ 7.1 Summary of Achievements

The original objectives of this thesis were:

1 To apply remodularisation to the hierarchy generation problem for state-based software model; and

2 To analyse and address qualitative issues with the output of existing automated remodularisation algorithms.

These aims were decomposed into three objectives:

1 Measure the application and analysis of an existing remodularisation technique to state-based software models

2 Produce enhancements to the state of the art of remodularisation by extending it to hierarchies

3 Develop and empirically evaluate an improved remodularisation approach through a realistic experiment with human participants

Figure 7.1 portrays the path of investigation followed by the thesis, directed by these two aims. Chapter 2 explored the problem domain, and Chapter 3 expanded and addressed Objective 1 it by examining the use of remodularisation to identify superstates for state machines. Three key problems were identified, the first two of which pertained to hierarchy generation and were further addressed in the work towards Objective 2 in Chapter 4. Chapters 5 and 6 addressed the final objective and concluded with a means to improve upon remodularisation algorithms using domain knowledge. Each of the contributions is elaborated in the following sections and further subsections.

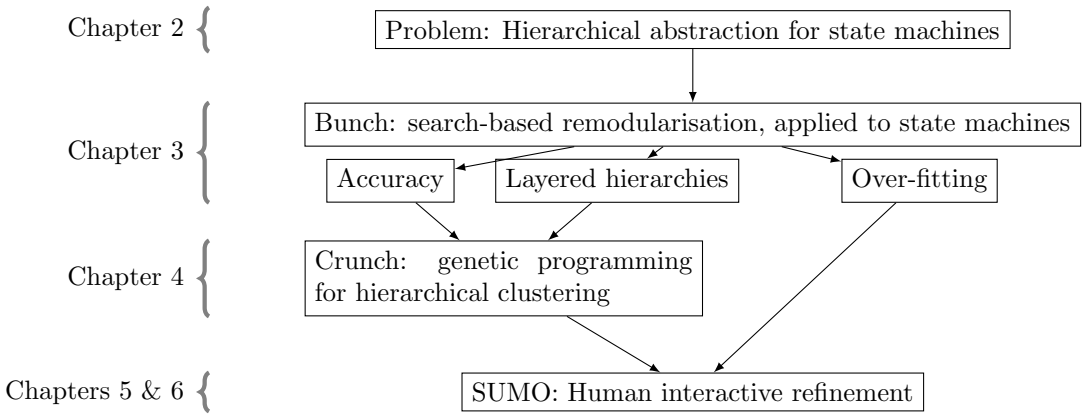


Figure 7.1: The path of investigation taken in the thesis

7.1.1 Remodularisation of State Machines

Chapter 3 showed that it was possible to apply the Bunch remodularisation algorithm to state-based models to produce superstates that resemble those produced by experts. The quality of the output produced by this technique was compared to known-good modularisations using established metrics, determining that, for the more complicated `tamagotchi` case study, Bunch did not entirely reproduce the original structure of the statechart. Although resultant modularisations were not completely correct, some included a large number of correct features.

The Bunch algorithm's performance was then explored, with a focus on its hierarchy generation approach. Investigation of several case studies showed that while unbalanced hierarchies are prevalent in existing systems, Bunch cannot produce them as builds a hierarchy in layers. Over-fitting caused by the layered approach to hierarchy formation was also identified.

Finally, a means of producing non-layered hierarchies was discussed, showing that the mean fitness of the clusters can be improved using a random deletion approach.

Chapter 3 demonstrated the feasibility of the use of Bunch for remodularisation of state machines, and identified three key problems:

Over-fitting in hierarchies The approach to hierarchy generation uses repeated searches, which allows the algorithm more freedom to explore solutions at the lower level than the higher levels

Non-layered hierarchies The layering approach to the search restricts the solutions to only those which merge clusters at each level, producing layered hierarchies; unlayered hierarchies were shown to be present in several open source projects

Accuracy The hierarchy for the larger of the two case studies was never successfully completely recovered by Bunch in the experiments.

7.1.2 Searching Hierarchies

In light of the first two problems identified in Chapter 3, an alternative search-based approach to remodularisation was introduced in Chapter 4. This approach uses representations that encode transformations of a hierarchy and label-based representations to optimise hierarchies at all levels in one search. Crunch was observed to produce higher fitnesses at higher levels of the hierarchy, at an expense of overall fitness over the whole hierarchy, using a novel adaptation of the MQ fitness function used in Bunch to evaluate hierarchies.

Various fitness functions were evaluated in Crunch with two state machine case studies. The types of structure produced by each fitness function were analysed, showing that two fitness functions from modularisation approaches did not produce solutions that allowed superstate edges to be produced. In contrast, the fitness functions that produced these superstate transitions also produced a greater number of clusters.

This latter evaluation highlighted a problem with automatic remodularisation approach that is widely acknowledged [12, 40, 81], that the quality of the output is difficult to assess, and is likely to be error-prone in circumstances where information beyond the structure of the graph is used by an expert to evaluate the quality of the result.

7.1.3 Domain Knowledge for Refinement

Chapter 5 showed that a refinement step could be used to address potential deficiencies of remodularisation algorithms by soliciting domain knowledge. In cases where clustering produced by a remodularisation algorithm is poor, an expert can bridge the gap to interactively produce an acceptable result using the SUMO algorithm, which encodes their domain knowledge as constraints. The number of constraints required is minimised through the use of a constraint solver.

The number of relations required for an arbitrary number of modules and classes was first derived theoretically. The typical number of relations required in a realistic scenario was then obtained through experimentation; the empirical evaluation used simulated authoritative decompositions [104] and measured the amount of effort (in terms of iterations) required to completely refine a modularisation, finding the number of required relations to be high. Crucially, the evaluation showed that SUMO produces measured improvement at almost each iteration, thus partial improvement can be gained if running the algorithm to convergence is infeasible.

Chapter 6 expanded on the evaluation of SUMO by evaluating its use with human subjects. This required an implementation of the algorithm with an interactive interface. A pilot study was conducted, the findings of which led to changes in the algorithm and the tool to increase its usability. The most significant of these changes was the pre-seeding of the constraint solver with the previous modularisation.

The findings from the empirical study in Chapter 6 showed firstly that the remodularisation problem does involve subjectivity; participants did not produce the same result even though the case study was identical for all participants. In terms of performance,

the study showed the feasibility of the use of the SUMO tool for remodularisation tasks, but found that the sigmoid curve observed in the study in Chapter 5 did not appear, nor did any other particular pattern beyond an apparent familiarisation period and a general trend of improvement over time. This demonstrated that the performance of the SUMO algorithm is highly dependent on the individual user.

7.1.4 Overall Conclusions

This thesis has demonstrated the similarity of module dependency graphs and state machines allow the Bunch remodularisation algorithm to be used to solve the state machine hierarchy generation problem to an extent. The user study performed in Chapter 6 has demonstrated that modularisation is a subjective process, showing that a fully automated approach is not a realistic goal.

The subjectivity of modularisation is problematic for automatic remodularisation algorithms as it proves that human intuition is necessary to ensure results are satisfactory to a user. While modularisation algorithms may continue to be improved through the use of better metrics and more accurate clustering techniques, the subjectivity issue will continue to be a problem. This issue is also acknowledged by Bavota et al., who have taken a similar approach to putting the developer "in the loop" with a remodularisation algorithm [12].

While domain knowledge is necessary, it is also important that the amount required is minimised as much as possible, as remodularisation has been automated to reduce the cost of reorganising a system. SUMO does this by ensuring that the user's information is applied as far as possible (through the transitive properties of the relation types and their application in a constraint solver). However, SUMO still requires a domain expert and a considerable investment in effort if it is to produce the best result possible.

This work represents an incremental improvement towards the practical combination of remodularisation to allow the improvement of both MDGs and state machines as software models in reverse engineering scenarios.

§ 7.2 Limitations and Future Work

This section outlines work that could further explore the remodularisation problem. It focuses on work which could follow from the two main topics investigated in this thesis; genetic hierarchical clustering (Crunch), and interactive modularisation refinement (SUMO).

7.2.1 Crunch

The work on Crunch in Chapter 4 demonstrated a potential for better quality hierarchies at a cost of overall fitness of the hierarchy, when compared to results from Bunch. These differences are possibly due to the effect of the use of the genetic programming technique used.

Alternative Representations and Extended Transformation Operators

Crunch is most critically bounded by the search space. While Bunch uses an agglomerative approach, where the search space is reduced at each increasing layer (until only one cluster is produced), Crunch does not use this approach and instead searches the space of all possible hierarchies. In addition to the search space size, the representations Crunch uses do not correspond to one-to-one mappings of individuals to solutions.

As it is possible to encode the same solution in multiple ways, this constitutes an artificial increase in the search space. Future work could include application of Harman et al. [46]'s alternative representation which does not have this problem. Another alternative would be the application of Seng et al.'s variable length chromosome approach [101] which does not require the individual to encode operations that have no effect that could cause a reduce in fitness when crossed over.

The design of the instruction set itself has a high impact on the resultant search space. Thus, further work on designing an instruction set or representation which has a low amount of duplication (causing artificial search space expansion) and is more resilient to problems of locality (the extreme example of which is the stack representation).

The two transformation operators implemented in Crunch cause it to produce a limited subset of all hierarchies. Work in refactoring exists that defines a greater set of transformations [101], some of which may be adaptable to the hierarchical clustering problem, such as the pulling up or down of attributes, which could correspond to elements in the clustering in this domain.

Mutation Operators for GP Clustering

As shown in the experiments, the stack-based implementation yielded poorer fitness values than the other representations, despite the stack-based implementation's ability to express more varied trees than the others. The extent to which this is dependent on the crossover could be investigated through the implementation of a tree-based clustering approach as used in Lutz [68]'s work on remodularisation. This could also allow performance of the other virtual machine clustering techniques.

Multiobjective Approaches

The experiments on the hierarchical performance of Crunch and Bunch showed that Bunch was able to better optimise the total MQ values of the hierarchy with its hill climbing approach than the global searches used by Crunch. Comparison of the depths of the hierarchies showed a large variation in the types of the hierarchies produced by the two tools, suggesting that they may be exploring different areas of the search space.

Objectively making a selection, even armed with these overall fitness values, is not possible however. The amount of layering is dependent on the case study in question; 5 levels may be reasonable in some cases, although it is perhaps reasonable to state

that 5 levels of hierarchy is inappropriate for a 6 state state machine. The choice of the criteria is dependent on expert knowledge unavailable to the search.

Multiobjective solutions to problems have been applied as a means to improve performance [94] as well as an approach to producing the best set of solutions that represents varying trade-offs of quality in various criteria, from which an expert can make a selection based on their judgement [48]. Such a multiobjective approach could remove the need to penalise Crunch for using an excessive amount of nesting (dictated by the penalty applied to it), by using the depth of the hierarchy as one of the objectives used in the search.

Interactive Evolution

Bavota et al. [12]'s recent success expands on a multiobjective approach with the incorporation of developer knowledge. The increased performance of this approach suggests that it is beneficial to provide a means for a user to guide the algorithm to a solution. Although Bunch [81] includes support for the incorporation of a user's ideas into the process, it does not allow the interactive process enabled by the IGA and the SUMO approaches.

Arch [100] exposes parameters of its algorithm to allow an expert to tune it to its application. A similar approach could be deployed in Crunch; by exposing the fitness function, or parameters to it, a user could encode features which they require that the fitness function did not already optimise for.

A key component of Bavota et al.'s approach is the use of the user-modified version of the individual to seed the population. Bunch itself also supports the seeding of the search with an existing starting configuration [81, 40]. This seeding approach could be implemented in Crunch by setting the initial configuration of the tree prior to transformation to the solution identified by the user. The "cost" of the refactoring, as used by Tonella in the evaluation of a concept-based approach to remodularisation [111] could be integrated into the fitness function to allow the user to express preference for changes are as similar to the original as possible.

Comparison of Performance

As Crunch is most similar to a hierarchical clustering algorithm, it could be compared with the performance of other hierarchical algorithms. This comparison could follow the same model of Anquetil et al. [9]'s analysis of hierarchical clustering algorithms by measuring the similarity of the algorithms at various cut heights.

In addition to analysis of the performance of Crunch through comparison with other algorithms, the results produced by Crunch could be evaluated using domain experts to determine how authoritative the modularisations it produces are.

Further Exploration of Fitness Functions

Experimentation with various fitness functions showed that Crunch provides a flexible environment for experimentation with hierarchical quality measures. However, the evaluation of a set of these measures in Section 4.5 proved difficult to interpret. Further evaluation of the hierarchies produced could be performed, replicating the experiments conducted in Chapter 3. However, the case studies used in this small case study are perhaps too small to warrant clustering at multiple layers—larger case studies would be required for such an experiment.

7.2.2 SUMO

The SUMO refinement algorithm provides a useful starting platform for introducing domain knowledge to the remodularisation process. Although it has been assessed in theoretical and empirical terms, the results show avenue for improvement. The fundamental performance measure of SUMO is the number of relations required, which in its evaluation has been found to be high for complete refinement. SUMO also only constructs a single partitioning of the graph, while other remodularisation approaches have aimed to produce hierarchies.

Although the SUMO tool has undergone evaluation and was improved through the pilot study, further enhancements to its function could be made. Firstly, the conflict resolution approach implemented of asking the user to correct their input (or use the undo command) could be expanded to determine which of the equalities the CSP solver found to be inconsistent to improve the usability of the tool.

Further work relates to methods to reduce of the amount of domain knowledge required for convergence. This could be potentially achieved through a variety of methods, detailed in the following subsections.

Solicitation of Constraints

The current implementation of SUMO gives the user freedom to supply any constraints they wish. This contrasts with Bavota et al. [12]'s interactive genetic algorithm, where the algorithm selects pairs of information that the user provides information for, based on their size or at random. This approach could be used to avoid the user supplying redundant information, for example, connecting all nodes in a cluster together.

This change from the direct incorporation of constraints to a constraint acquisition problem may allow algorithms such as CONACQ [15] to be applied. CONACQ differs from SUMO in that it aims to acquire the set of constraints through the use of *membership* queries. CONACQ works by asking the user to categorise several examples as either positive or negative. From these classifications, CONACQ determines the set of constraints over them based on the input. The number of queries required is reduced by careful selection of examples for classification.

Non-binary Relations

SUMO uses binary constraints for the feedback mechanism, however participants in the pilot study (discussed in Section 6.3.5) remarked that the relation type was difficult to determine: classes appearing to be tangentially related presented a problem, as neither of the options of "together" or "not together" were appropriate.

Using fuzzy logic instead of these Boolean constraints could alleviate this problem. The user would be able to encode their certainty that two elements should be together. This could also be combined with a machine learning algorithm, or another remodularisation algorithm, to provide an estimate of similarity based on the user's perception, rather than the apparent similarities present in the data set.

Numeric values for the constraints, rather than Boolean values, could also be used to form distance values, similar to those used in hierarchical clustering. This could permit SUMO to construct hierarchies, although this would potentially increase the quantity of information required from the user, for example, Likert values would be required in place of the current yes/no binary relation type.

Collaborative Remodularisation

The experiments conducted with SUMO show that it is a costly process, however a necessary one, given the disagreement between the participants' produced modularisation and that produced by Bunch. The individual cost of the SUMO algorithm could be reduced by allowing multiple users to supply their domain knowledge. This change would allow larger systems to be remodularised without the requirement for the user to reverse engineer parts of the system they may be unfamiliar with. Conflicts between users could be managed using the previously suggested fuzzy logic implementation.

§ 7.3 Final Remarks

This thesis has investigated the remodularisation approach from several sides. In addition to contributing solutions and assessment of those solutions, it also identifies several difficulties inherent to the challenge presented by remodularisation.

Remodularisation is difficult for a number of reasons, confounded by the difficulty in efficiently searching all solutions. This thesis focused on algorithms that are flexible and scalable in the sense that they can be run for a budgeted amount of time to produce an increase in quality if sub-optimality is acceptable. SUMO in particular represents a practical approach, as it allows the user to interactively explore the system and modularise it, as demonstrated in the user study in Chapter 6.

This work represents a step towards an interactive solution, however there are still further avenues to explore to further reduce the burden on the human.

Bibliography

[1] JRET: Java Reverse Engineering Toolkit. `https://wiki.cse.yorku.ca/project/cluster/jret`.

[2] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 1998.

[3] S. Ali, K. Bogdanov, and N. Walkinshaw. A comparative study of methods for dynamic reverse-engineering of state models. Technical Report CS-07-16, Department of Computer Science, University of Sheffield, 2007.

[4] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178. Springer, 1999.

[5] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In J. Launchbury and J. C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 4–16. ACM, 2002.

[6] 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*, pages 79–88. IEEE Computer Society, 2005.

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

[8] N. Anquetil and T. Lethbridge. Extracting concepts from file names: A new file clustering criterion. In K. Torii, K. Futatsugi, and R. A. Kemmerer, editors, *Forging New Links, Proceedings of the 1998 International Conference on*

Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998, pages 84–93. IEEE Computer Society, 1998.

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

[10] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1–10. ACM, 2011.

[11] V. Batagelj and A. Mrvar. Pajek-program for large network analysis. *Connections*, 21(2):47–57, 1998.

[12] 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 G. Fraser and J. T. de Souza, editors, *Search Based Software Engineering - 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings*, volume 7515 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2012.

[13] C. Becker, S. Glomb, and M. Graf. UML Notation and Ilogix Rhapsody Tool. Seminar on Tool-supported modeling of Tamagotchi, University of Kaiserslautern, 1998.

[14] P. Berkhin. A survey of clustering data mining techniques. *Grouping Multidimensional Data*, pages 25—71, Jan 2006.

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

[16] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Transactions on Computers*, C-21(6):592–597, 1972.

[17] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.

[18] K.-T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In A. E. Dunlop, editor, *DAC '93 Proceedings of the 30th international Design Automation Conference*, pages 86–91. ACM, 1993.

[19] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[20] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[21] M. Chis. Evolutionary Hierarchical Clustering Techniques. *Acta Universitatis Apulensis*, 2(4):57–64, 2002.

[22] M. Chis. Hierarchical Clustering Using Evolutionary Algorithms. In G. Felici and C. Vercellis, editors, *Mathematical Methods for Knowledge Discovery and Data Mining*, pages 146–156. IGI Global, 2008.

[23] H. Chu, Q. Li, S. Hu, and P. Chen. An Approach for Reversely Generating Hierarchical UML Statechart Diagrams. In L. Wang, L. Jiao, G. Shi, X. Li, and J. Liu, editors, *Fuzzy Systems and Knowledge Discovery, Third International Conference, FSKD 2006, Xi'an, China, September 24-28, 2006, Proceedings*, volume 4223 of *Lecture Notes in Computer Science*, pages 434–437. Springer, 2006.

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

[25] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering Methodology*, 7(3):215–249, 1998.

[26] S. Counsell, S. Swift, A. Tucker, and E. Mendes. Object-oriented cohesion subjectivity amongst experienced and novice developers: an empirical study. *ACM SIGSOFT Software Engineering Notes*, 31(5):1–10, Sept. 2006.

[27] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 1985*, pages 183–187. Lawrence Erlbaum Associates, 1985.

[28] C. Damas, B. Lambeau, P. Dupont, and A. v. Lamsweerde. Generating Annotated Behavior Models from End-User Scenarios. *IEEE Transactions on Software Engineering*, 31:1056–1073, 2005.

[29] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions Evolutionary Computation*, 6(2):182–197, 2002.

[30] C. Di Francescomarino, A. Marchetto, and P. Tonella. Cluster-based modularization of processes recovered from web applications. *Journal of Software Maintenance and Evolution: Research and Practice*, Sept. 2010.

[31] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of java dependency graphs. In R. Koschke, C. D. Hundhausen, and A. Telea, editors, *Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008*, pages 91–94. ACM, 2008.

[32] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science*, 276(1):51–81, 2002.

[33] P. Dupont, B. Lambeau, C. Damas, and A. van Lamsweerde. The qsm algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22(1&2):77–115, Jan 2008.

[34] I. D. Falco, E. Tarantino, A. D. Cioppa, and F. Gagliardi. A novel grammar-based genetic programming approach to clustering. In H. Haddad, L. M. Liebrock, A. Omicini, and R. L. Wainwright, editors, *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, pages 928–932. ACM, 2005.

[35] N. E. Fenton and A. Melton. Deriving structurally based software measures. *Journal of Systems and Software*, 12(3):177–187, 1990.

[36] C. Ferreira. Gene expression programming: a new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[37] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2):139–172, 1987.

[38] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[39] M. Genero, D. Miranda, and M. Piattini. Defining metrics for uml statechart diagrams in a methodological way. In M. A. Jeusfeld and O. Pastor, editors, *ER (Workshops)*, volume 2814 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[40] M. Glorie, A. Zaidman, L. Hofland, and A. van Deursen. Splitting a large software archive for easing future software evolution - an industrial experience report using formal concept analysis. In *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece*, pages 153–162. IEEE, 2008.

[41] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification: Java SE 7 Edition*. Oracle, July 2012.

[42] M. Hall. Search based hierarchy generation for reverse engineered state machines. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 392–395. ACM, 2011.

[43] 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*, pages 472–481. IEEE Computer Society, 2012.

[44] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[45] M. Harman. The current state and future of search based software engineering. In L. C. Briand and A. L. Wolf, editors, *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 342–357, 2007.

[46] M. Harman, R. M. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In W. B. Langdon, E. Cantú-Paz, K. E. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. K. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9-13 July 2002*, pages 1351–1358. Morgan Kaufmann, 2002.

[47] M. Harman, S. Swift, and K. Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1029–1036. ACM, 2005.

[48] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In H. Lipson, editor, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2007, London, England, UK, July 7-11, 2007*, pages 1106–1113. ACM, 2007.

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

[50] M. Holcombe. X-machines as a basis for dynamic system specification. *Software Engineering Journal*, 3(2):69–76, 1988.

[51] J. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[52] E. R. Hruschka, R. J. G. B. Campello, A. A. Freitas, and A. C. P. L. F. de Carvalho. A Survey of Evolutionary Algorithms for Clustering. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 39(2):133–155, Mar. 2009.

[53] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, 1985.

[54] JUNG Contributors. JUNG - Java Universal Network/Graph Framework. `http://jung.sourceforge.net/index.html`.

[55] N. Jussien, G. Rochart, and X. Lorca. The choco constraint programming solver. In *CPAIOR08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP08)*, 2008.

[56] A. Kalaji, R. Hierons, and S. Swift. An Integrated Search-Based Approach for Automatic Testing from Extended Finite State Machine (EFSM) Models. *Information and Software Technology*, 53(12):1297–1318, 2011.

[57] J.-P. Katoen. Labelled transition systems. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 615–616. Springer, 2004.

[58] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Series in Probability and Statistics. Wiley, 1990.

[59] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In J. L. Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[60] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, June 1994.

[61] R. Krovi. Genetic algorithms for clustering: a preliminary investigation. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 4, pages 540–544, 1992.

[62] A. Kumar. SCHAEM: A Method to Extract Statechart Representation of FSMs. In *Proceedings of the IEEE International Advance Computing Conference (IACC)*, pages 1556–1561. IEEE International, Mar. 2009.

[63] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In V. Honavar and G. Slutzki, editors, *Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings*, volume 1433 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[64] T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, 2003.

[65] D. Lo and S.-C. Khoo. Quark: Empirical assessment of automaton-based specification miners. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 51–60. IEEE Computer Society, 2006.

[66] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 501–510. ACM, 2008.

[67] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis

tools with dynamic instrumentation. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 190–200. ACM, 2005.

[68] R. Lutz. Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture*, 47(7):613—634, Jan 2001.

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

[70] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*, pages 315–324. IEEE Computer Society, 2003.

[71] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy*, pages 45–. IEEE Computer Society, 1998.

[72] O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[73] A. Marchetto, C. D. Francescomarino, and P. Tonella. Optimizing the trade-off between complexity and conformance in process reduction. In M. B. Cohen and M. Ó. Cinnéide, editors, *SSBSE*, volume 6956 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2011.

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

[75] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308—320, Jan 1976.

[76] K. Meffert et al. JGAP - Java Genetic Algorithms and Genetic Programming Package. `http://jgap.sf.net`.

[77] B. Mitchell and S. Mancoridis. On the evaluation of the Bunch search-based software modularization algorithm. *Soft Computing*, 12(1):77–93, 2008.

[78] B. S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD thesis, Drexel University, 2002.

[79] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of*

the IEEE International Conference on Software Maintenance, pages 744–753, 2001.

[80] B. S. Mitchell and S. Mancoridis. Craft: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 93–, Washington, DC, USA, 2001. IEEE Computer Society.

[81] B. S. Mitchell and S. Mancoridis. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[82] M. Mitchell, J. H. Holland, and S. Forrest. When will a genetic algorithm outperform hill climbing? *Advances in neural information processing systems*, pages 51–51, 1994.

[83] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse-engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, dec 1993.

[84] R. Naseem, O. Maqbool, and S. Muhammad. Improved similarity measures for software clustering. In T. Mens, Y. Kanellopoulos, and A. Winter, editors, *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*, pages 45–54. IEEE Computer Society, 2011.

[85] Object Management Group. *Unified Modeling Language Specification (version 2.1)*, 2007.

[86] M. K. O'Keeffe and M. Ó. Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, Apr. 2008.

[87] J. Oncina and P. Garcia. Identifying Regular Languages In Polynomial Time. *Advances in Structural and Syntactic Pattern Recognition*, 5:99–108, 1992.

[88] M. Orlov and M. Sipper. Flight of the FINCH Through the Java Wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, 2011.

[89] D. L. Parnas. Software aging. In B. Fadini, L. J. Osterweil, and A. van Lamsweerde, editors, *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*, pages 279–287. IEEE Computer Society / ACM Press, 1994.

[90] M. D. Penta, M. Neteler, G. Antoniol, and E. Merlo. A language-independent software renovation framework. *Journal of Systems and Software*, 77(3):225–240, 2005.

[91] T. Perkis. Stack-Based Genetic Programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation, 1994*, pages 148–153, 1994.

[92] G. D. Plotkin. *A Structural Approach to Operational Semantics*, Jan 1981.

[93] E. Poll and A. Schubert. Verifying an implementation of SSH. In *WITS 2007: 17th Annual Workshop on Information Technologies & Systems*, pages 164–177, 2007.

[94] K. Praditwong, M. Harman, and X. Yao. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

[95] O. Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.

[96] W. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, Jan 1971.

[97] J. Rissanen. Modeling by shortest data description. *Automatica*, 14, Jan 1978.

[98] M. E. Romera. Using Finite Automata to Represent Mental Models. Master's thesis, San Jose State University, 2000.

[99] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall Series in Artificial Intelligence. Prentice Hall/Pearson Education, 2003.

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

[101] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In M. Cattolico, editor, *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006*, pages 1909–1916. ACM, 2006.

[102] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[103] M. Shtern and V. Tzerpos. A framework for the comparison of nested software decompositions. In *11th Working Conference on Reverse Engineering (WCRE 2004), 8-12 November 2004, Delft, The Netherlands*, pages 284–292. IEEE Computer Society, 2004.

[104] M. Shtern and V. Tzerpos. Evaluating software clustering using multiple simulated authoritative decompositions. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 353–361. IEEE, 2011.

[105] M. Shtern and V. Tzerpos. Clustering methodologies for software engineering. *Advances in Software Engineering*, 2012.

[106] SourceNav NG Development Group. Source Nagivator NG. `http://sourcenav.berlios.de`, 2012.

[107] J. Sun and J. Dong. Design Synthesis from Interaction and State-based Specifications. *IEEE Transactions on Software Engineering*, 32(6):349–364, 2006.

[108] T. Systa, K. Koskimies, and E. Makinen. Automated compression of state machines using UML statechart diagram notation. *Information and Software Technology*, 44(10):565–578, 2002.

[109] J. Tessier. Dependency finder. `http://depfind.sourceforge.net`, 2012.

[110] The C++ Standards Committee. *Working draft, Standard for Programming Language C++*. ISO/IEC, 2012.

[111] P. Tonella. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering*, 27(4):351–363, 2001.

[112] V. Tzerpos and R. C. Holt. A hybrid process for recovering software architecture. In M. A. Bauer, K. Bennet, W. M. Gentleman, J. H. Johnson, K. A. Lyons, and J. Slonim, editors, *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative Research, November 12-14, 1996, Toronto, Ontario, Canada*, page 38. IBM, 1996.

[113] V. Tzerpos and R. C. Holt. The Orphan Adoption problem in architecture maintenance. In *Proceedings of the Fourth Working Conference on Reverse Engineering, 1997.*, pages 76–, 1997.

[114] V. Tzerpos and R. C. Holt. MoJo: a distance metric for software clusterings. *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 187–193, 1999.

[115] V. Tzerpos and R. C. Holt. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267. IEEE Computer Society, 2000.

[116] V. Tzerpos and R. C. Holt. On the stability of software clustering algorithms. In *8th International Workshop on Program Comprehension (IWPC 2000), 10-11 June 2000, Limerick, Ireland*, pages 211–. IEEE Computer Society, 2000.

[117] W. van der Aalst, H. A. Reijers, and A. Weijters. Business process mining: An industrial application. *Information Systems*, 2007.

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

[119] K. Wagstaff and C. Cardie. Clustering with instance-level constraints. In P. Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 1103–1110. Morgan Kaufmann, 2000.

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

[121] N. Walkinshaw and K. Bogdanov. Applying Grammar Inference Principles to Dynamic Analysis. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis*, pages 18–23, 2007.

[122] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In M. D. Penta and J. I. Maletic, editors, *14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada*, pages 209–218. IEEE Computer Society, 2007.

[123] N. Walkinshaw, J. Derrick, and Q. Guo. Iterative refinement of reverse-engineered models by model-based testing. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2009.

[124] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *12th International Workshop on Program Comprehension (IWPC 2004), 24-26 June 2004, Bari, Italy*, pages 194–203. IEEE Computer Society, 2004.

[125] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering, 1997*, pages 33–43, 1997.

[126] 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*, pages 525–535. IEEE Computer Society, 2005.

[127] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Feb. 1979.

Appendix A

§ A.1 Help Sheet Used in the Experiment

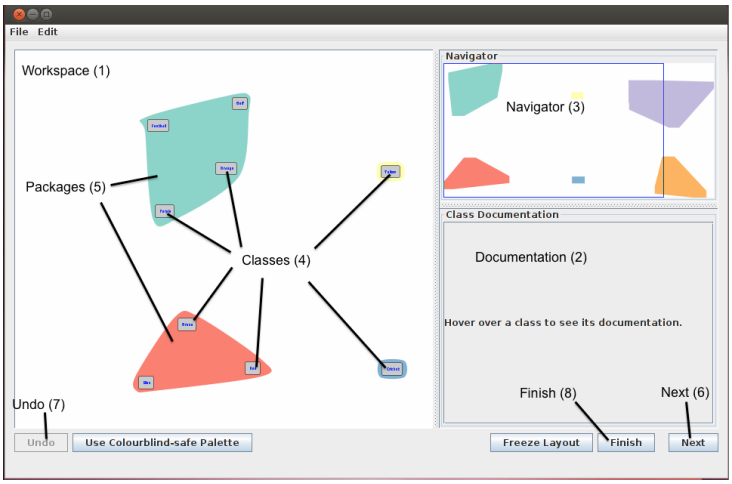


Figure 1: The goal of the exrcise is to use the tool to improve the set of packages (5) for the classes (4). The documentation (2) will help you make your choices.

Moving around (1) and (3)

- Drag the left mouse button on a white region in the Workspace (1) to move around
- Drag the right mouse button, or use the scroll wheel to zoom in or out
- The Navigator (3) can also be used to pan the workspace

Adding Information (4)

To add information between classes:

Click the first item This will select it and turn it yellow. Then:

Left click Creates a green edge. The elements will always be together whenever Next is pressed.

Right click Creates a red edge. The elements will never be together from the next time you click Next.

Making changes

You can change coloured links by right clicking on them and selecting an item from the menu that appears. You can't change the greyed out edges this way — the Undo button must be used instead.

Locking packages

To tell the tool to not make any changes to a package, right click it then select the "Lock module" option. You can only undo this operation using the Undo (7) button.

Refining the packages (6)

Pressing the Next button (6) will make a new set of packages that match the information you supply. This finalises the links between the classes, **they can't be changed in future steps without using the Undo button**.

Undo (7)

The Undo button (7) returns you to the previous solution. Any edges which are not greyed out will be lost. This might be useful if the solution you get when pressing Next isn't what you want, or if you want to go back to remove some already saved (shown in grey) links.

Finishing (8)

Once you're happy with the set of packages and can't see any more changes that should be made, press Finish. You'll then add some extra information to your packages: a short description and your confidence in how good the module is, where 5 is best and 1 is the worst.