Superstate Identification for State Machines Using Search-Based Clustering

Mathew Hall
m.hall@dcs.shef.ac.uk

Phil McMinn
p.mcminn@dcs.shef.ac.uk

Neil Walkinshaw
n.walkinshaw@dcs.shef.ac.uk

Department of Computer Science
University of Sheffield, Regent Court, 211 Portobello
Sheffield, S1 4DP

ABSTRACT

State machines are a popular method of representing a system at a high level of abstraction that enables developers to gain an overview of the system they represent and quickly understand it.

Several techniques have been developed to reverse engineer state machines from software, so as to produce a concise and up-to-date document of how a system works. However, the machines that are recovered are usually flat and contain a large number of states. This means that the abstract picture they are supposed to provide is often itself very complex, requiring effort to understand.

This paper proposes the use of search-based clustering as a means of overcoming this problem. Clustering state machines opens up the possibility of recovering the structural hierarchy of a state machine, such that superstates may be identified. An evaluation study is performed using the Bunch search-based clustering tool, which demonstrates the usefulness of the approach.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation, reverse engineering*

General Terms

Algorithms, design, experimentation

Keywords

State machines, search-based clustering, hill-climbing, Bunch

1. INTRODUCTION

Over long periods of development and maintenance on a piece of software, specification and design documents inevitably become neglected, and diverge from the actual operation and structure of the system. When the software requires bug fixes or the implementation of further features, these documents can potentially do more harm than good, should they even still be in existence.

In such circumstances, the developer needs to invest valuable time familiarising or re-familiarising themselves with the code of the system, building up an abstract picture themselves of how a system works and how its components relate to one another.

One way of lowering the cost and human effort involved in performing this task is to use automatic techniques that can reverse engineer design documents through static and dynamic analysis [6].

Reverse engineering of state machines from software, has received much interest [1, 8, 20, 21]. However the machines synthesised tend to be flat and lacking a hierarchical structure, consisting of many states; meaning that the derived machines are themselves complex and hard to understand.

This paper addresses the problem of rediscovering the hierarchical structure of a state machine, through analysis of its transition structure. The approach taken is to cluster the states of the machine so that related states are grouped together. In this way, the potential superstates of the machine may be revealed. The expectation is that a hierarchical state machine will allow a developer to grasp more quickly the behaviour of a software system.

The clustering method used is the search-based method of Mitchell and Mancoridis [16], which has been implemented in a tool called Bunch. Bunch was originally used to cluster module dependency graphs, but has since been adapted to other clustering problems in the software engineering field, including dynamic clustering of the heap in order to improve the performance of garbage collection algorithms [7].

This is the first work in which Bunch is applied to clustering state machines, with the aim of deriving its hierarchical structure. An evaluation study is performed to two case studies to assess its usefulness, with promising results.

The contributions of this paper are therefore:

1. The application of the search-based clustering tool Bunch to clustering state machines into a two-level hierarchy, so that the potential superstates of the machine are identified

2. An evaluation study that applies Bunch to two flattened state machines, with an assessment of how close the clustering technique is to recovering the original 2-level hierarchical structure of the original machine

This paper is organised as follows. Section 2 introduces some background to clustering, the application of search-based techniques to the problem, and the Bunch tool. Section 3 discusses the application of Bunch to clustering state machines. Section 4 then presents the evaluation study, with Section 5 presenting and discussing the results obtained. Section 6 presents related work, while Section 7 concludes and details future work.

2. BACKGROUND - CLUSTERING

Unsupervised or automatic clustering is a mature problem in the fields of Computer Science and Statistics. Despite this maturity, there still does not exist a general clustering algorithm; this is unlikely to change due to the computational complexity of the problem. Automatic clustering seeks to produce logical groups of a set of entities where each entity is similar to other entities in a subset, but significantly different to those from other subsets. The fundamental problem with producing a general clustering algorithm is this measure of similarity; often it is difficult to generalise the difference between two entities as a single distance measure.

Popular metrics used typically include some numerical value that gauges the similarity between objects, such as the euclidean distance. While this is not an issue when the data to be clustered is numeric or can be represented numerically it becomes a problem for more complicated data.

One of the most popular methods used in this area is k-means, an application of Lloyd's algorithm; wherein k data points are assigned randomly or via a heuristic, these become the 'means'. Each data point is assigned to the cluster with the closest mean. After the initial clustering is performed the centroid of each cluster is obtained and data points moved to each cluster based on the proximity to the origin of each centroid. This iterative process continues until the clustering converges. Although effective, k-means is also very costly as it is iterative and determining the correct value of k is difficult. The k-means algorithm is NP-hard so its utility is somewhat restricted and additionally it may not converge on the global optimum.

Another problem with clustering algorithms is the size of the data set; for large sets of elements exhaustively searching for the best clustering is infeasible in this domain [15]. The problem is made particularly worse when the number of clusters is not defined. Because of this problem, a search-based approach is used.

Search-based Clustering

By using a search algorithm to find the optimum solution large state machines can be considered rather than only those for which exhaustive search is feasible. The clustering method detailed in this paper is built upon work performed by Mitchell and Mancoridis in software module dependency graphs (MDGs) [16]. In their research they applied an agglomerative approach to clustering these dependency graphs; where the solution is developed by merging clusters, starting from an initial state where each cluster contains one element. A search-based approach was used; they concluded that hill-climbing was superior to a genetic algorithm approach [16].

Mitchell and Mancoridis produced a tool called Bunch to implement the technique developed in their research. The fitness function used, called 'Modularisation Quality' (MQ), rewards clusterings that have a low number of edges between clusters in proportion to the edges within the cluster. The MQ value for k clusters is given by:

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

$$CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum\limits_{\substack{j=1 \\ j\neq i}}^{k} (\epsilon_{i,j} + \epsilon_{j,i})} & \text{otherwise} \end{cases}$$

Where μ_i is the number of edges within a cluster (intra edges) and $\epsilon_{i,j}$ is the number of edges from cluster i to cluster j (the inter edges).

3. CLUSTERING OF FINITE STATE MACHINES

The original intention behind Bunch was to cluster module dependency graphs, which are directed graphs, where nodes represent modules and edges are dependencies. Since state machines are also directed graphs, with nodes as states and edges as transitions from one state to another, state machines may also be clustered by the tool without the need for any adaptation.

One difference in applying Bunch, however, centres around *reflexive edges* in the graph. These are removed by Bunch prior to clustering, because for module dependency analysis (Bunch's original design goal), they add no new information. For state machines, however, a reflexive edge is a transition that leaves and re-enters the same state. Thus although they will play little part in the overall structure of the machine, they at least need to be re-instated after the clustering process so as to produce an equivalent machine to the original. In addition, transition labels are not catered for by Bunch, and so these are also reinstated after the clustering process.

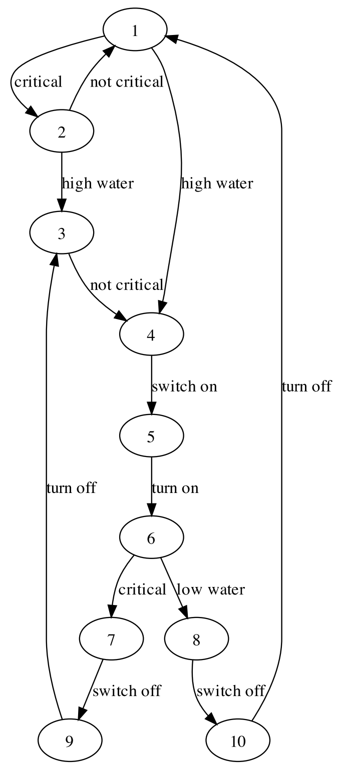
Figure 1a depicts a state machine for a water pump controller [9]. The clustering by Bunch is depicted in part b, showing two superstates that correspond to states where the water level is high and low.

Of course, the reduction in complexity of the hierarchical machine is dependent on the clustering making sense. As the clustering is driven by structure of nodes and edges alone, the clustering may not necessarily correlate with what a human would deem to be logical. The extent to which this is the case using this technique is examined in the next section.

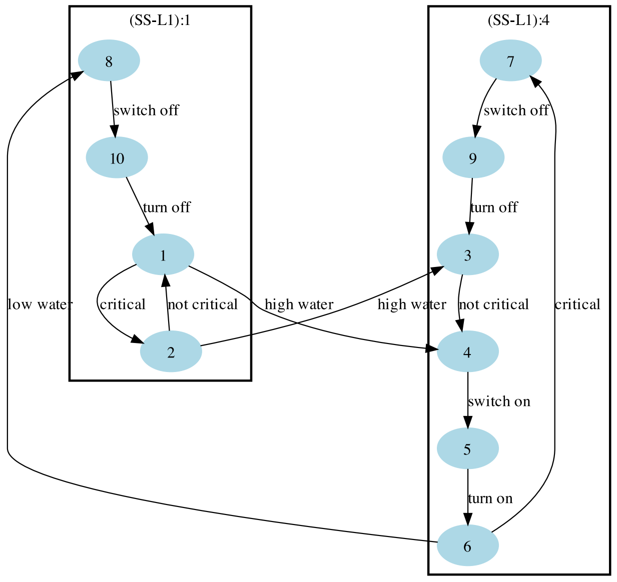
4. METHOD OF EVALUATION

In order to evaluate the ability of Bunch to recover superstate information, two statecharts were taken and flattened. Bunch was then run on the example 30 times, with the results compared back to the original hierarchical machine. As Bunch uses a hill-climbing approach (as with any search based technique) there is a possibility of it converging on a local optimum; the repeated runs are used in an attempt to reduce the probability that results consist entirely of such suboptimal solutions.

The two flattened state machine case studies are shown in Figures 2a and 3a respectively. The first example state machine is an alarm clock. The original statechart consists of 3 superstates; the statechart contains superstates for normal operations (show the time, display the alarm time and



(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 1: The original and clustered state machine for a water pump controller

beep) and operations for setting the alarm time, as well as the current time.

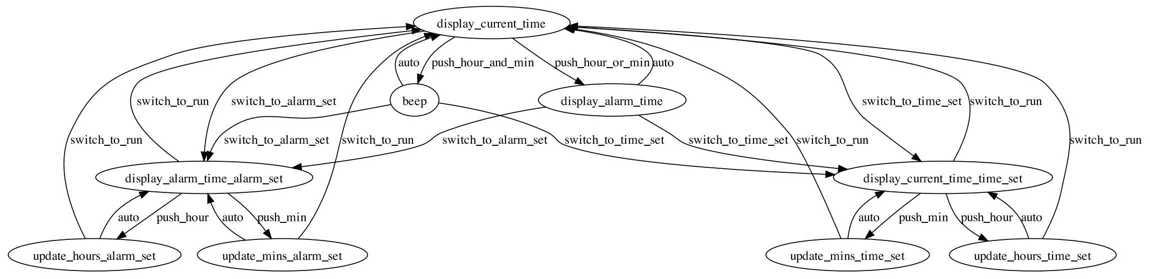
The second machine is taken from a student project to produce UML documentation for a 'tamagotchi' electronic pet toy. The super states for this machine include a main menu, which has a sub-state for showing a menu of actions, a food menu with states for meal or snack, a status viewer with states for each statistic of the state of the pet *weight, nutrition, happiness, weight_age*, and a catch-all playing sub-state consisting of a single *playing* state. In addition to these states there are two states at the top level for idle and showing the clock.

Each statechart was flattened, such that a state machine containing all the nodes in the statechart was produced. To flatten a machine the superstates are removed; this is performed by replacing each transition from the superstate with a transitions for each substate. For example for a superstate S consisting of states $\{s_1, s_2, s_3\}$ where there is a transition in the statechart $S \rightarrow f$, the transitions $s_1 \rightarrow f$, $s_2 \rightarrow f$ and $s_3 \rightarrow f$ are produced. At present this is a manual process but incorporation this functionality into a tool for easier experimentation is intended. This collapsing method makes the assumption that a state that is not part of a substate should be placed in its own cluster by itself.

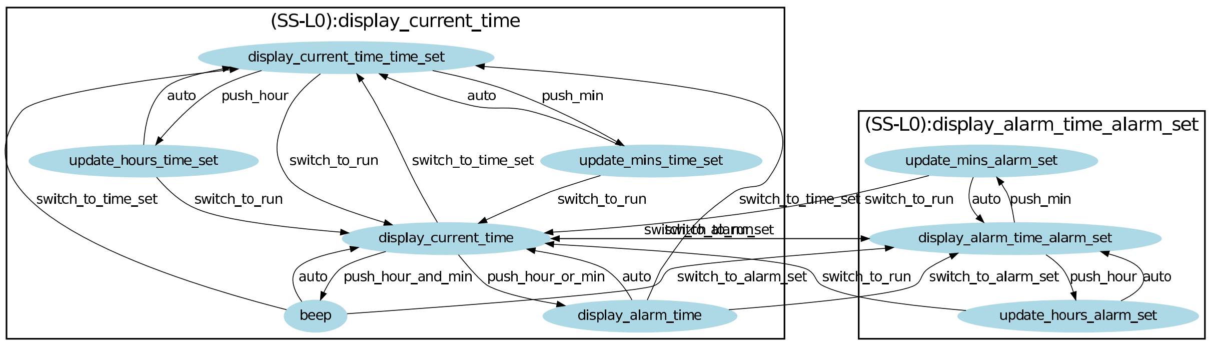
Where the original statechart involved concurrent states, the largest state was chosen as the statechart in its own right, since concurrency is not accounted for in simple directed graph state machines. The statecharts were also assumed to operate without memory; that is, when a transition occurs between superstates it always goes to the start state of the superstate rather than any previous states from it. Conflicting state names were resolved by adding the name of the superstate they were obtained from, as otherwise they would be treated as equal. As this technique ignores state labels this changing of state names does not interfere with the clustering in any way.

The flattened state machines were then run through Bunch a total of 30 times using the default agglomerative hill-climbing algorithm using the weighted incremental MQ objective function. The comparisons included recording the number of clusters that Bunch generated that also appeared exactly in the statechart as well as the EdgeSim and MeCl metrics developed by Mitchell [15]. Both EdgeSim and MeCl are designed to quantify the similarity of clustered graphs. EdgeSim (Edge Similarity) aggregates the weight of each of the edges in two clusterings of a graph if the edge is the same type in each, intra or inter. MeCl characterises the difference between one clustering and another by counting the number of inter edges introduced after join operations required to produce one clustering from the other have been performed.

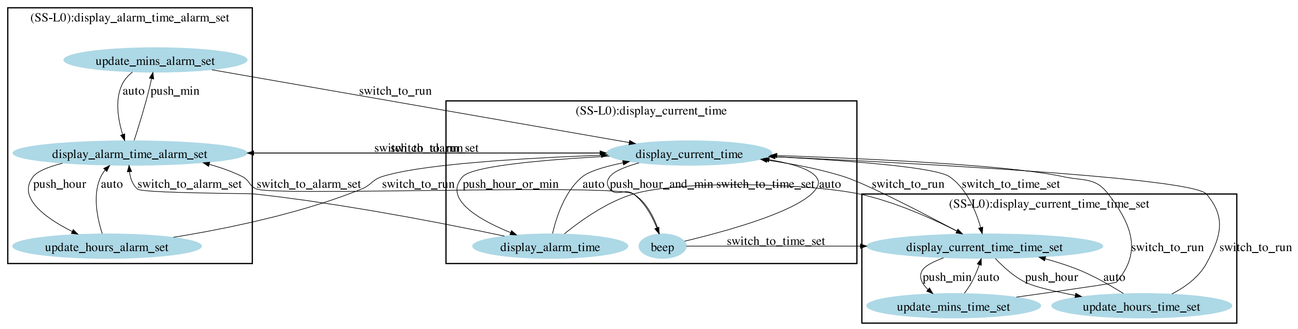
The metrics were obtained by running the calculator from Bunch on the output, using the statechart as a comparison. Precision and Recall results for each clustering produced were also obtained in the same way as Bunch provides a component to calculate these. In the case where multiple levels were produced the evaluation was performed on each level of clustering.



(a) The flattened state machine for the alarm clock statechart of Romera [18]

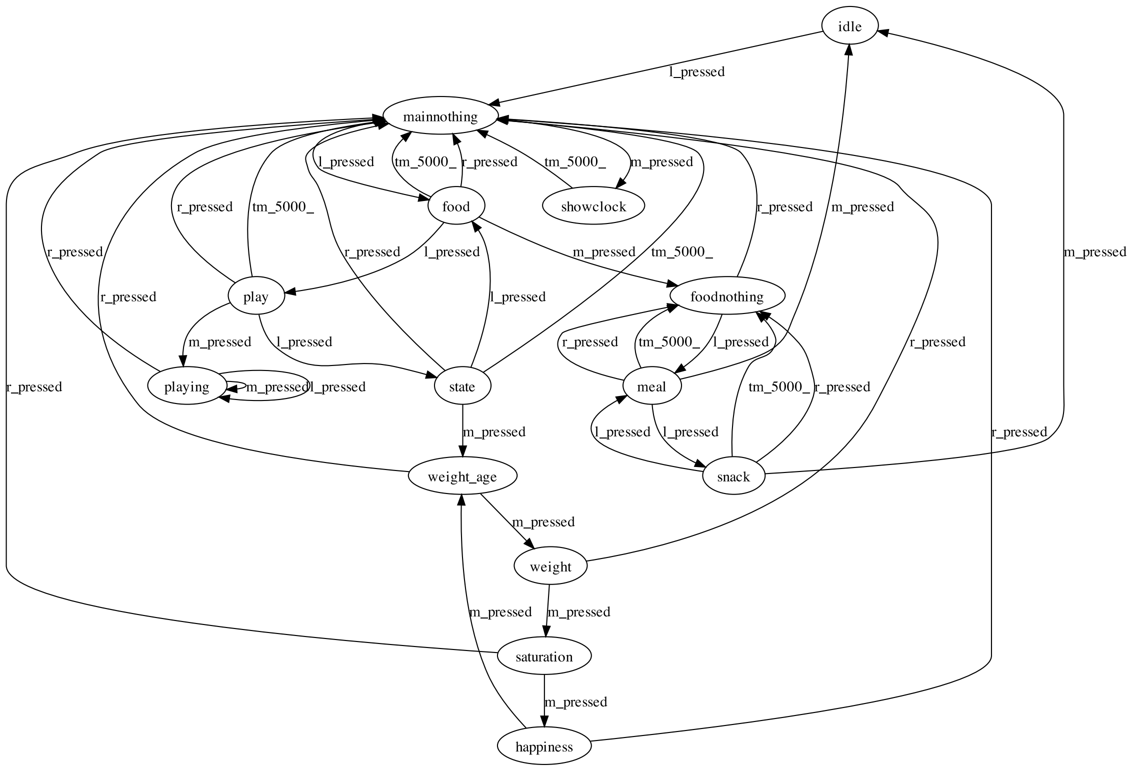


(b) A clustered state machine from one run of Bunch (MQ value 1.39)

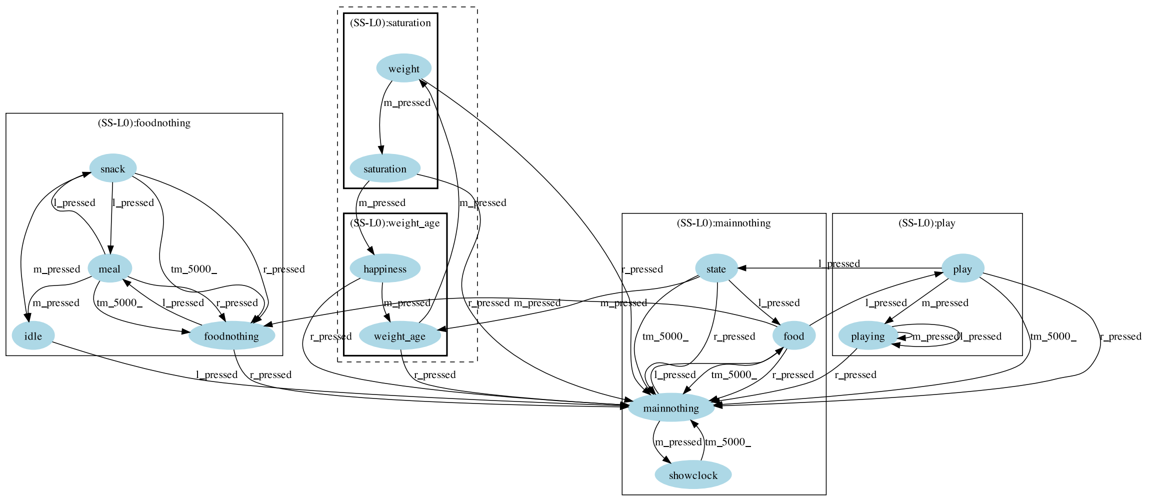


(c) A clustering that produced superstates identical to the original statechart, with an MQ of 1.54

Figure 2: The alarm clock case study



(a) The flattened state machine for the 'tamagotchi' statechart of Becker *et al.* [3]



(b) An example clustering produced by Bunch. The superstates identified by the clustering are all but identical to those of the original statechart, except that two super states are merged in the original (the dashed line, manually added, indicates the superstate that was further divided by Bunch)

Figure 3: The 'tamagotchi' case study

5. RESULTS

The results are shown below in Tables 1 and 2 for the alarm clock and tamagotchi case studies respectively. The 'States as in statechart' column refers to the number of states that were placed in the same cluster as in the target clustering. The perfect score here corresponds to the number of states in the machine, *i.e.* 9 for the alarm clock and 14 for the tamagotchi case study.

As shown by the tables of results the first machine clustered with a high degree of success; routinely scoring 100% for all metrics. Although MQ and accuracy correlate for the first machine this is not the case with the second machine; higher EdgeSim and MeCl values are associated with lower MQ scores. This suggests that using MQ to attempt to recover the clusters from statecharts is not possible without a high degree of error.

Results for the tamagotchi machine were better with the MeCl and EdgeSim metrics at the detail level (level 0) than the median level (level 1); although more clusters were correct at the median level. The "level" mentioned is a Bunch-specific term — the "median" and "detail" levels are likely to be the most interesting when more than 2 levels of clustering are produced; this is particularly relevant for large state machines that may produce many levels.

When working at the median level for the tamagotchi case study, the 'State' substate was clustered according to the sample clustering from the statechart; however this was divided into two clusters at the lower level, shown in figure 3 — this is reflected in the statechart as the other states are one level deeper in terms of a hierarchy.

The tamagotchi case study included two reflexive edges; these are removed by Bunch as discussed in section 3 and as such this may have skewed the clustering away from the anticipated result. As well as the reflexive edges the second machine was far less uniform than the first; this structure does not conform to the low coupling and high cohesion that is expected in the MQ fitness function.

Although the numeric results obtained do not show a high degree of accuracy for the more complicated 'real world' 'tamagotchi' machine the clusterings produced do exhibit some degree of promise. Similarly named states were placed together and in the case of the "state" superstate the clustering was exactly as in the statechart, although at the detail level this cluster was subdivided. This suggests that the technique may be viable despite the fact that it does not accurately recover the hierarchy of a statechart; its application may lie in recommending possibly related states as an aid in reverse engineering or perhaps even forward engineering.

The evaluation method was intended to be as objective as possible; in practice this could not be the case as ultimately the statecharts used cannot be considered "perfect" as even their own design is subjective to the person who created them. As the quality of the evaluation is dependent on the correctness of the statecharts used it is not possible to make a confident conclusion as to the viability of this technique on the whole. A superior evaluation method would include inviting human subjects to produce statecharts for a machine and determining if the technique duplicates any of them, as well as investigating the extent to which human subjects agree with a statechart representation proposed by the technique. Essentially this technique's evaluation is invested in a degree of uncertainty as it is impossible to purely objectively evaluate its performance; but the area it is aimed at, system design, is also subjective to a degree.

6. RELATED WORK

Search-based clustering has been developed by many authors [16, 11, 15] to recover the structure and relationship between modules in a piece of software. This is the first work to apply clustering to state machines.

A variety of approaches have been proposed for dealing with state machines that are too large or complex for their intended tasks. These can be broadly divided into two camps. One approach is to adopt more powerful modelling approaches (e.g. Harel Statecharts and Extended Finite State Machines (EFSMs)) that incorporate abstraction mechanisms to hide the underlying complexity. The other approach is to develop analysis techniques such as slicing and clustering to help the developer home-in on sections of the machine that are of particular interest.

A range of more powerful abstract state-based modelling techniques have been developed to address the complexity problem. Notations such as EFSMs [5] or Abstract State Machines [4] augment conventional finite state machines by incorporating a memory and labelling transitions with abstract functions on that memory. This enables a potentially complex range of behaviours to be represented in a much more compact format.

Hierarchical state machines, such as the UML variants of Harel Statecharts [10] also include the notion of nested states. They not only permit abstract transition functions, but also incorporate more complex semantics to account for high-level states that contain their own nested state machines. Whenever a super-state is reached, its nested state machine represents the ensuing behaviour within that state.

The modelling approaches listed above are effective at representing complex state-based behaviour and have to a greater or lesser extent been adopted in practice. However, even if such modelling approaches are adopted, there are two problems: (1) Identifying appropriate abstractions can be extremely challenging and time-consuming, and (2) the resulting model can still be too complex to be of use. The first problem (at least partially) addressed by the technique proposed in this paper; it removes the burden from the developer by *automatically* proposing state machine abstractions. The second problem is elaborated below.

The technique proposed in this paper is not the first approach to address the important problem of state machine abstraction. Two basic approaches exist that attempt to do so: slicing, which takes advantage of variable information in extended finite state machines (see above) and clustering, which analyses the state transition structure.

Slicing was originally developed as a source code analysis technique [22], which could be employed to identify portions of the source code that could affect the behaviour of a given statement. A slice is computed by a combined analysis of control and data-flow. Over the past decade, a growing body of work has considered the application of slicing techniques to extended finite state machines [12, 2]. The rationale is that a slice would isolate only those states and transitions in the EFSM that are related to the behaviour of a particular state.

There has been little empirical work to demonstrate the efficacy of slicing approaches, which makes them difficult to compare with the work presented in this paper. There seems

	Depth	MQ	Clusters	EdgeSim	MeCl	Prec.	Rec.	States as in statechart	MQ Evals
Avg.	6.43	1.47	2.53	97.5	99.8	76.67	100	6.2	143.73
Min	2	1.39	2	87.5	99	50	100	3	111
Max	11	1.54		100	100	100	100	9	182
Std Dev.	2.51	0.08	0.51	5.09	0.41	25.37	0	3.04	22.63

Table 1: Overall results for the alarm clock state machine

	Depth	MQ	Clusters	EdgeSim	MeCl	Prec.	Rec.	States as in statechart	MQ Evals
					Level 1				
Avg	3.93	1.35	2.23	51.3	80.3	29.11	100	4	-
Min	0	1.27	2	46.66	79	23.8	100	4	-
Max	9	1.39	3	66.66	88	45.55	100	4	-
Std Dev	2.53	0.06	0.43	8.56	3.09	9.2	0	0	-
					Level 0				
Avg	14.7	2.27	5.87	58.97	89.33	48.18	40	0	574.2
Min	8	2.26	5	50	85	36.36	40	0	387
Max	23	2.28	6	69	90	50	40	0	861
Std Dev	3.98	0	0.35	3.93	1.73	4.71	0	0	115.61

Table 2: Results for both levels of clustering of the 'tamagotchi' state machine

to be a complementary relationship between the two techniques. Clustering techniques are 'coarser', requiring only a state transition structure, with no interaction from the developer. Slicing techniques on the other hand work at a lower level, requiring the developer to supply specific slicing criteria. The authors intend to explore this relationship in more detail in their future work.

To the best of the authors' knowledge, the only other paper to propose an automated technique for grouping similar states together was proposed by Huawei *et al.* [14]. As with the slicing techniques mentioned above, they assume that the state machine incorporates data constraints and is a more-or-less complete description of the underlying software behaviour. They group states together by analysing the data constraints that govern state transitions. The clustering approach adopted in this paper is a departure from their line of work because it does not rely on data constraints. It can be applied to any state machine, whether it is abstracted or not.

This work is closely related to the domain of the StateChum state machine inference tool [19]. StateChum implements a variety of algorithms designed to produce a finite state machine from a set of traces through a program. StateChum is able to produce machines that are more representative of the program as they do not include any unreachable states; this comes at a price however, without an exhaustive set of test data the machine may exclude some interesting states. In addition to this the process is quite intensive as minimising the machine requires sufficient information to deduce that states are equivalent in order to merge them.

7. CONCLUSIONS AND FUTURE WORK

This paper investigated the application of the Bunch search-based clustering tool to the clustering of states in a state machine. For large state machines, clustering may identify higher level 'superstates', which help developers to quickly understand state machine diagrams representing a system. The method of applying the Bunch tool to the problem was evaluated with two case studies. In both cases, promising, but not always perfect, results were obtained.

The possibility of incorporating other information into the clustering process remains to be investigated. There is a possibility that state labels may be usable as an approximated distance metric, perhaps through application of the Levenshtein distance metric [13]. By approximating a distance metric between states it is anticipated that it will be possible to apply graph clustering algorithms as suggested by Rattigan et al. [17]. There are other properties of the graph that may be usable as a distance metric, including the number of transitions in common between nodes for example; each of these may be usable to at least aid in clustering a machine. It is also possible to conclude that MQ may not be best suited as a fitness function and could be augmented or replaced with a more appropriate metric, such as EVM as used by Harman et al. [11].

Of particular interest is the performance of this technique when contrasted with related techniques. StateChum's grammar inference process is quite intensive. Although this technique performs clustering it may be applicable to the domain of grammar inference as a grammar can be represented by a state machine. Since this technique is concerned on the structure of a state machine only rather than the actual meaning of transitions it runs in a very small amount of time. It may be the case that this technique can be used in conjunction with StateChum by providing suggestions of states to merge; those that are clustered together may be better merge candidates.

The technique appeared to recover multiple levels of a state hierarchy from the collapsed statechart which suggests it may be applicable in production of hierarchies rather than single clusterings. Although this is ideal as it would allow generation of more useful diagrams for larger systems it also creates a problem in that when clustering at one level some clusters encapsulate nodes that should be in different clusters and at a lower level clusters higher in the hierarchy are divided into smaller sub-clusters which are not accurate when compared with the target clustering.

The long term road map for this research is to produce a stand-alone tool that is able to perform hierarchical clustering driven by a number of algorithms, search-based or otherwise. As the prime focus is on producing a tool that will operate on output from StateChum, the effect it has on the clustering of machines will be examined as it can introduce noise. The initial development of the tool will be concerned with replicating the functionality of Bunch in a manner that enables additional information (such as labels) as well as reflexive edges to be used in evaluating a candidate clustering.

Acknowledgements

The authors would like to thank Spiros Mancoridis and Brian Mitchell for providing the Bunch source code, without which this research would not have been possible. Mathew Hall and Neil Walkinshaw are funded through the EPSRC project REGI.

8. REFERENCES

[1] G. Ammons, R. Bodík, and J. Larus. Mining Specifications. In *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 4–16, Portland, Oregon, 2002.

[2] K. Androutsopoulos, N. Gold, M. Harman, Z. Li, and L. Tratt. A theoretical and empirical study of EFSM dependence. In *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, pages 287–296. IEEE, 2009.

[3] C. Becker, S. Glomb, and M. Graf. Uml notation and ilogix rhapsody tool. Seminar on Tool-supported modeling of Tamagotchi, University of Kaiserslautern, 1998.

[4] E. Börger. Abstract state machines and high-level system design and analysis. *Theoretical Computer Science*, 336(2-3):205–207, 2005.

[5] K. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*, pages 86–91, 1993.

[6] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[7] M. Cohen, S. B. Kooi, and W. Srisa-an. Clustering the heap in multi-threaded applications for improved garbage collection. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1901–1908, New York, NY, USA, 2006. ACM.

[8] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[9] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31:1056–1073, 2005.

[10] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[11] M. Harman, S. Swift, and K. Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Genetic and Evolutionary Computation Conference*, 2005.

[12] M. Heimdahl and M. Whalen. Reduction and Slicing of Hierarchical State Machines. In *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering (FSE'97)*, 1997.

[13] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics-Doklady No 8*, 10:707–710, 1966.

[14] H. Li, Y. Min, and Z. Li. Clustering of behavioral phases in FSMs and its applications to VLSI test. *Science in China Series F: Information Sciences 45(6), 462-478. (2002)*, 2002.

[15] B. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD thesis, Drexel Universit, Philadelphia, PA, 2002.

[16] B. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. In *IEEE Transactions on Software Engineering*, 2006.

[17] M. J. Rattigan, M. Maier, D. Jensen, B. Wu, X. Pei, J. Tan, and Y. Wang. Exploiting network structure for active inference in collective classification. In *ICDM Workshops*, pages 429–434, 2007.

[18] M. E. Romera. Using finite automata to represent mental models. Master's thesis, San Jose State University, 2000.

[19] N. Walkinshaw. Statechum project website.

[20] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse Engineering State Machines by Interactive Grammar Inference. In *14th IEEE International Working Conference on Reverse Engineering (WCRE)*, 2007.

[21] N. Walkinshaw, J. Derrick, and Q. Guo. Iterative refinement of reverse-engineered models by model-based testing. In *Proceedings of FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2009.

[22] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.