A comparative study of methods for dynamic reverse-engineering of state models

Shaukat Ali
The Software Quality Engineering Laboratory,
Carleton University, Canada
Email:
shaukat@sce.carleton.ca

Kirill Bogdanov
Dept. of Computer Science,
University of Sheffield, UK
Email:
K.Bogdanov@dcs.shef.ac.uk

Neil Walkinshaw
Dept. of Computer Science,
University of Sheffield, UK
Email:
N.Walkinshaw@dcs.shef.ac.uk

Abstract

This paper provides a comparison of dynamic analysis techniques to reverse engineer state models of software. Since running a program generates a lot of data, the utility of different techniques is related to their ability to construct models which are both reflecting the relevant characteristics of a program and of manageable size. The classification of techniques in this paper aims to highlight the different approaches to abstraction utilised in the surveyed papers, so that one can mix-and-match different techniques depending on the kind of models which need to be built.

1 Introduction

Software is often available without a specification adequate for program analysis or test generation; this may be due to development starting with only an informal specification or the original specification not being maintained when a program is changed. In addition, a specification adequate for some tasks may not be for others; for instance, in situations when one is solely concerned with ensuring that racing conditions do not occur between multiple threads, it could be enough to check that locking obeys specific rules [28]. On the contrary, testing of the behaviour may require a different type of a specification, such as statecharts [25]. This paper surveys different techniques for using dynamic analysis to construct state-based models of software at a chosen level of abstraction, shows how the techniques can be compared and how one may attempt to combine ideas from some of them to synthesize new methods of dynamic analysis. The goal is to survey methods to construct state-based models of software.

Testing or program understanding [11] are not the only use for such a specification - one may use it to model-check software or stub parts of a system which are either not implemented or not relevant for the analysis being performed. The idea of using a specification in place of a part of a system is used in a number of papers, such as [34, 43, 42]; such a specification does not have to be complete, but it has to be enough for the purpose of analysis or test generation. Construction of system models is different to many existing software model-checking [23, 20, 33] approaches because model-checking only builds models to the extent necessary to check a property; model construction stops once either no counter-example is found or the one found is exhibited by a program being checked.

2 The choice of papers surveyed

Papers surveyed have been chosen to extract control-related behaviour and feature a broad range of ideas what to call a state and a transition label in a system; this survey aims to highlight different approaches to reverse engineering rather than provide a semi-complete collection of papers on the subject. For this reason, papers utilising similar techniques to those described below were not included.

A model extracted from code can be represented in many different forms, almost as many as there are specification notations. In this paper, the emphasis is on state-based models, which are useful for analysis and test generation [25, 38]. Explicit preconditions and operations associated with transitions are useful to represent data transformations, but difficult to handle during automated test generation, because having constructed paths through a model, one has to find inputs to make a system follow those paths. In a similar

way, it may be difficult to analyse a model heavily reliant on conditions. Software model-checking papers mentioned above solve this problem by only considering those conditions which are relevant to the property being checked; for building a model, one would want to represent as much as possible on a transition diagram, so that either a model-checker or powerful state-machine -derived test methods can be applied to it. For this reason, most methods mentioned below extract labelled transition systems (LTS), potentially with annotations [13] which can help generate test data during test generation. Such annotations are generated [13] by the Daikon tool [32] to identify constraints on arguments of object method calls. Daikon has also been used to build models of software [41, 37], but assertions it extracts reflect constraints on data; none of the papers surveyed use it to reverse-engineer state machines.

It is well known that static analysis techniques (such as [27, 47, 36]) complement dynamic ones; for this reason (1) many of the techniques surveyed below have a corresponding symbolic execution equivalents and (2) the two types of analysis are often used in conjunction [8, 45, 26]. Static analysis can find all paths, but is usually conservative in that not all of the discovered paths can be executed; dynamic analysis is effective in generation of paths which are known to be executable, but will not necessarily generate all executable paths. Even when one can obtain all paths, there would be an infinite number of them and most of those paths would not be of interest to a developer building a model, hence one has to construct an abstract model which preserves properties he/she is interested in. Abstraction can be applied at different stages in model construction, such as at collection of traces where not all traces are considered or at model-building where certain states are considered equivalent for a modelling purpose.

3 Dynamic analysis model

This section starts with a description of the overall pattern followed by papers on dynamic analysis. It subsequently focuses on specific parts of this model, each described in the individual subsection.

This paper attempts to show how numerous papers can be shown to fit roughly the same pattern of dynamic analysis: one would explore a program by running it on a number of tests, collect data from executions and subsequently build a model. This is depicted in a data-flow diagram presented in Figure 1. The diagram is similar to models presented in [15, 5] but does not include multiple abstraction stages.

The *program execution* box corresponds to running a program on a set of tests or real data and collecting information. A 'program' could be a single class under test, a cluster of classes or an entire system — it appears possible to classify techniques in a similar way regardless of the level at which they are applied, however approaches to abstraction and exploration would clearly be very different between unit- and system-level reverse-engineering. Most techniques operate at a unit level (Table 1); exceptions include [8] which builds dynamic models related to a single field of an object and [9, 13] which build system-level models.

Abstraction level	Technique using it
Object field	the dynamic analysis part of [8]
Object/unit	[1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 15]
Subsystem/ system	[9, 13, 14, 16, 17]

Table 1. Levels of abstraction of state models

Execution of a program to reverse-engineer usually involves instrumenting it, such as using byte code instrumentation for Java or source-code instrumentation. Low-level traces obtained from executions may contain all events sent or received by a program and all method calls it received or made; variable values may also be included. The *trace abstraction* box corresponds to (1) filtering out irrelevant information as defined by an expert user (mentioned in [35] and other papers), and (2) and constructing an abstract representation of traces. In the simplest form, this may involve only keeping method names and specific variable values from detailed traces; in a more complex case, one may replace method arguments with generic names (to suggest that a particular trace does not depend on those arguments). The analysis of different techniques presented below highlights approaches to abstraction taken by different authors. *Model building* corresponds to the construction of models from abstract traces; this may involve simply unifying abstracted method calls and variable values, alternatively, learning techniques such as those surveyed in [40, 29] can be used. The outcome of learning can be shown to a user for analysis or it can be used for test generation.

The three *feedback* arrows indicate different kinds of feedback possible in the described approach to reverse-engineering: one may explore a system at a low level, such as making sure that specific variables have taken on all interesting values or at a systems level [35] where traces extracted from a constructed model are used to validate the model (and check the system for bugs).

The summary of the main results of this paper is presented in Table 2. It includes the type of model

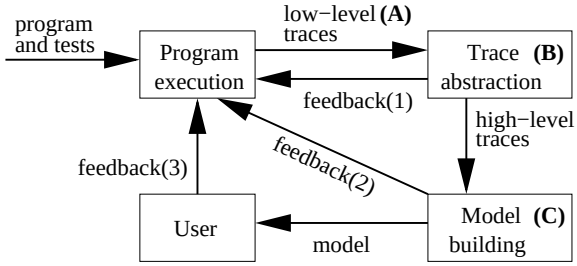

Figure 1. Dynamic analysis

built by each reference, the low-level information used (column), a way to abstract it (column **(B)**) and a model building method used (column **(C)**). The last column reflects the type of feedback used. Numbers after feedback names reflect the level of abstraction at which feedback is used and correspond to numbers on *feedback* transitions in Figure 1. The column corresponding to low-level information is split into two parts, corresponding to state and label-related data; the column devoted to abstraction is split into state- and label-related abstraction methods.

Every technique mentioned in a table is described in a subsection corresponding to the column of the table. Many of the techniques are hard to understand without knowing the context in which they are applied by the respective papers. For this reason, under the heading corresponding to each of techniques, the relevant part of a respective paper is briefly summarised.

3.1 Models built

LTS Labelled-transition systems describe systems containing a fixed number of states; transitions between states are labelled with events or method calls. When a system is in a specific state and an event is received, a corresponding transition is taken to a new state. Consider, for instance, a stack shown in Figure 2. The initial state is *Empty*, a call of `Push` will get the stack to state *One Element*. In the initial state, it is not possible to call `Pop` or `Top`; it is not possible to call `Push` in the *Full* state either. Compared to any real implementation, this particular model does not describe how data is handled, indeed, this model only ensures that the number of calls to `Push` is greater or equal than the number of calls to `Pop`, but does not exceed it by more than 3. This is an example of abstraction mentioned above, where elements of a system not considered relevant (in this case, the data stored in the stack) are ignored.

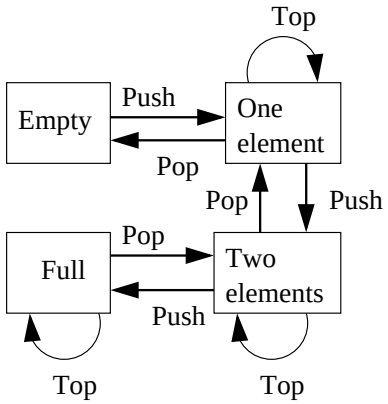

Figure 2. A stack which can hold up to three elements

Regular expressions Any sequence of method calls which does not involve calls prohibited from certain states (such as `Pop` from the initial state in Figure 2) can be followed in the above-mentioned LTS. In contrast, finite-state acceptors have a notion of *accept*-states; only sequences of calls leading such an acceptor to an accept state are allowed. Acceptors are more general than LTSs in that they may allow a sequence of calls such that no prefix of such a sequence is allowed; in contrast, if an LTS allows a sequence, it allowes all of its prefixes. Regular expressions correspond exactly to finite-state acceptors; this is a classical result in finite-state automata theory.

Methods as states Reference [8] introduces models where every state corresponds to a method and transitions are unlabelled; transitions connect pairs of methods, one of which may follow another one.

Markov Markov chains are effectively LTSs with probabilities. Every label is has a probability of it being taken associated with it. For instance, when running a system for a while, one may record the number of times each transition of the above stack has been taken. Subsequently, one can annotate transitions with probabilities. For instance, the likelyhood that a `Push` will be called from the *Two elements* state may be lower than a `Pop` from that state.

IOTS Reference [9] collects low-level trace information in a telephone network, such as messages exchanged or the lights which are lit; no state-related information is obtained at this stage. After abstraction, an iterative learner algorithm is used to build an

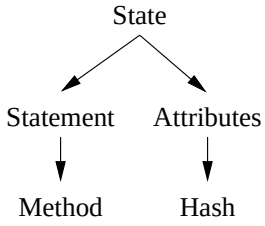

Figure 3. Low level states

I/O transition system; the result of learning is different from that in LTS learning papers because the there could be multiple outputs produced per single input. This is the reason why an *input/output transition system (IOTS)* is built by [9] where labels represent either an input or an output.

Algebraic Algebraic specifications describe software using a notion of axioms, relating both source and target states and data passed between a system and its user, for instance `Push(5).Push(6).Pop() = Push(5)` or `Push(5).Push(6).Top() = 6`. Axioms are equations describing the behaviour or every method call in every state. Methods are split into *constructors*, *modifiers* and *observers*. Sequences of the former methods describe what is needed to get a system to enter every state (such as `Push(5)`), modifiers are methods one considers from every state and observers are methods used to distinguish between states. One expects constructors and modifiers to modify a state of an object and observers not to. In the example above, the first axiom shows that a call of `Push(6).Pop()` in the `Push(5)` state does not change a state. The second axiom states that a call of an observer `Top()` in the `Push(5).Push(6)` state returns 6.

3.2 Low-level trace information

Papers that are described here start by running an object or a system through a number of tests, collecting information from its execution. Information collected to form a state of a future model can be either *statement*-related or *attribute*-related. In the former case, states in models are program statements, such as branch statements [14] or method calls [8, 10]. In the latter, values of member variables of an object or a system are observed.

WholeState, Calls-Args, frontier, orig states Papers [1, 2, 4, 5, 7] can use the Rostra [46] tool for bounded state-space exploration. Rostra uses either a random or a commercial test set generator both for the initial state-space exploration and for generation of arguments to methods. Method calls attempted from all states during this initial stage are collected and subsequently attempted from all reached implementation states. These states are identified with attribute values of an object under test and all objects it refers to (recursively); this is is referred to as the *WholeState* way of state identification. *Orig states* below corresponds to attempting all collected calls from all implementation states reached by the original test set; methods called and argument values are recorded (*Calls-Args*). *Frontier* is a breadth-first exploration of a concrete state space, with the maximal number of times the frontier is moved defined by a user. It is worth noting that Rostra [46] supports multiple different state equivalences, only one of which is actually used in the surveyed papers.

Hash Reference [6] performs a bounded state space exploration, where reached states are identified with *hash*es of object variables.

Statement, Likelyhood that one state follows another one Paper [14] builds a classifier to predict the outcome of program behaviour based on the branches taken during that execution. This makes it possible to run a program on a number of tests and record for each test (1) how often every branch is taken and (2) oracle classification of this test. A specific number or a sequence of numbers are possible oracle outputs. At the low level, one collects the number of times each branch is taken, with a record for each test; this corresponds to a Markov chain which is a state machine with states being program statements and transitions between a pair of states labelled with the number of times control passed from one of these states to another one. After normalisation, one can view any of these weights on transitions as a probability that in a given test execution a particular statement will follow another one.

3.3 Abstraction of traces

EQN Paper [6] can abstract states using [30]'s observational equivalence. In the framework of algebraic specification, a model of an object corresponds to a number of axioms, describing an equivalence of two sequences of method calls executed on the said object. In the course of state-space exploration, an object may reach a state visited before (with a different sequence of method calls); in this situation the two call sequences are considered equivalent. Observer methods are those which do not modify an object but report on

some aspect of its state, such methods may be useful to distinguish states; a more fine-grained way is to attempt state-modifying methods, followed by an observer. Two concrete states are considered observationally equivalent if any sequence of method calls return the same value; in reality, one has to put a bound on the length of such method sequences. Reference [6] can determine if a pair of concrete states are equivalent by attempting a number of method sequences and checking if they return the same value. In a similar way, terms corresponding to (parts of) method arguments in an axiom can be abstracted: it is enough to replace such terms with a number of different terms and check the axiom being abstracted for validity.

Daikon-like In contrast to [6], [7] learns axioms by performing bounded state-space exploration and then checking which axioms (from a given set of templates) are satisfied. In a sense, this is what Daikon [32] does; in comparison, [7] uses algebraic axioms and generates tests rather than relies on a user to supply them.

In a similar way to the Daikon-like learning mentioned above, [17] learns temporal properties by using templates of them and checking how often they are satisfied when a given system runs through the tests. This way, it is possible to learn universally-true properties. Given that in addition to templates the relative generality of properties is provided (which property implies another one), [17] also learns more general properties which hold only on certain parts of the extracted traces.

Papers [1, 2, 3, 4, 5] do not generally perform an abstraction of argument values, but can filter them out for presentation purpose [2]. These papers use three different types of state abstraction.

Branches The *branches* abstraction corresponds to attempting a pre-selected set of method calls from the same concrete state and collecting branch coverage per call in the set; one may think of this information as a map from the chosen calls to decisions made by the object under test. Concrete states which produce the same maps are consider equivalent. In some sense, this is similar to state abstraction of [6], but uses white-box (coverage) information instead of black-box (observational equivalence).

Variable slices The *variable slices* abstraction corresponds to a selection of a number of attributes of an object and using their values to distinguish between states; for attributes pointing to other objects, one may use *structural abstraction* to consider such attributes equivalent if reference attributes of the objects they point to are in turn equivalent. In a sense, *structural abstraction* corresponds to WholeState where primitive attributes are ignored.

Observers The *observers* abstraction uses multiple observer methods to report on the state; two different concrete states are considered equivalent if all observers report the same values when invoked from each of these states.

Paper [5] constructs high-level state machines by building low-level ones using any of the branches, variable slices or observers abstractions and subsequently constructing a cross-product of them. Label abstraction involves keeping only external calls to objects within a considered cluster of objects and annotating arguments to these calls with constraints. The aim of the annotations serves primarily to ensure that only results of previous calls to the cluster can be passed as method arguments (this is conceptually similar to grouping constraints in [27]).

Filtering Paper [15] abstracts traces using two techniques, filtering and *standardization*. The former relies on define-use dependencies between states provided by a user and seed states which select subsets of low-level traces for abstraction. For a user-defined constant N, filtering selects at most N states on which the seed state depends and at most N states dependent on the seed one, with a few extra states which both depend on some chosen states and affect others.

Standardization The *standardization* abstraction [15] replaces specific values of variables by generic constants; this seems similar to how one might abstract timestamps in [9].

Roles Objects with the same values of member variables may play different roles [12] if, for instance, one of them is a leaf of some tree and another one is not. A number of roles of an object are defined on the basis of referencing relationships. For instance, if an object is not reachable, then special role garbage is reserved for this purpose; identity relationship determines whether two paths within a data structure lead to the same object. By default, roles are computed at each method entry and exit; one can make methods atomic so as to avoid recording role transitions associated with calls that those methods make. If is also possible to associate roles taken by an object with a different object; this is useful when complex data structures are implemented using nested containers, permitting one to associate roles taken by a lower-level container with a higher-level one.

Ref.	Model	Low-level traces **(A)**		Abstraction applied to get high-level traces **(B)**		model building **(C)**	feedback
		state	label	state	label		
[1]	LTS	WholeState	Calls-Args	branches	–	equivalence	frontier(1)
[2]	LTS	WholeState	Calls-Args	variable slices	method signature	equivalence	frontier(1)
[3]	LTS	WholeState	Calls-Args	observers	–	equivalence	orig states(1)
[4]	LTS	WholeState	Calls-Args	observers	external calls and dependencies	equivalence	frontier(1) and tests from the model(3)
[5]	LTS	WholeState	Calls-Args	branches, variable slices, observers	–	equivalence	frontier(1)
[6]	Algebraic	hash	Calls-Args	hash/EQN	EQN	equivalence	–
[7]	Algebraic	WholeState	Calls-Args	–	–	Daikon-like	frontier(1)
[8]	Methods as states	method signature	external calls	–	–	equivalence	–
[9]	IOTS	–	external events	–	filtering, standardization	Angluin, partial order, symmetry	Angluin queries(3)
[10]	Regular expressions	–	method signature	–	–	clustering(2)	–
[11]	LTS	method signature	method signatures and branches taken	–	–	Method of [24]	–
[12]	LTS	WholeState	method signature	Roles	–	equivalence	–
[13]	LTS	–	Calls-Args	–	equivalence + Daikon	GK-Tails	–
[14]	Markov	Statement	Likelyhood that one state follows another one	Statement	clustering(1)	–	asking a user for unknown traces
[15]	LTS	Calls-Args	–	dependency-filtering, standardization	–	K-tails (1)	–
[16]	LTS	–	events	–	–	K-tails (2)	–
[16]	Markov	–	events	–	–	Markov learner	–
[17]	Regular expressions	–	events	–	–	Daikon-like	–

Table 2. The summary of the comparison between different methods of reverse-engineering via dynamic analysis

Equivalence + Daikon Reference [13] forms an abstraction by merging traces corresponding to the same sequences of method calls and subsequently uses the Daikon tool to infer constraints on argument values supplied to those calls.

Clustering(1) Paper [14] abstracts low-level branch information by replacing per-test Markov chains with aggregate information, built by selecting subsets of sets of chains for each outcome of an oracle and adding together weights on transitions (with subsequent normalisation). The decision which models to aggregate is based on a measure of distance between them. Merging is performed iteratively; at each step, one (1) merges a pair of 'close' models and (2) stops when all models close to nearby ones have been merged. The stopping criterion is a sharp rise in a standard deviation of the difference between models. The trace-related part of the entry in Table 2 corresponding to [14] reflects a single aggregate model.

3.4 Model building

Equivalence Many methods that are surveyed in this paper build models by utilising the equivalences over the abstracted states and labels, so that an abstract model consists of abstracted states and the labels on transitions between them correspond to labels between them in high-level traces.

Clustering(2) Reference [10] uses clustering to build regular expressions corresponding to API usage extracted at run-time. Clustering affects sequences of method calls which are similar; similarity is based on the number of textual insertions/deletions/substitutions necessary to rewrite one sequence into another one. After that, sequences corresponding to each cluster are rewritten in a form of regular expressions, encompassing all sequences in a cluster. For instance, if all sequences in a cluster start with a number of method calls `A`, an abstract regular expression could start with $\texttt{A}^+$.

Angluin Model-building of [9] reflects usage of both a modification of the Angluin algorithm for an interactive learner [19] and expert input. The learner is one of a family of regular grammar inference techniques [40, 29], it attempts to build a transition system not only using traces initially offered to it but also by constructing queries and using their outcomes to update a model. The idea of learning can be summarised as follows: traces correspond to paths through a system, but states they traverse are not directly visible, so one could attempt to construct longer and longer traces to visit all traces in a system and attempt to distinguish between states based on traces which are possible from them. One can hence identify a correspondence between learning and testing techniques [21]. The Angluin algorithm builds a table where rows are labelled with traces visiting states and columns — traces used to distinguish between states. Every cell in a table corresponds to a specific row and a column; it contains a response of a state corresponding to a row, to a sequence of inputs corresponding to a column. Angluin uses just yes/no answers, [9] extends this to sequences of outputs. In the described way, for every state one may determine how it behaves in response to a number of sequences of inputs. If one takes an input from this state, it would usually lead to another state. Calling this state A, one might introduce another row corresponding to it and obtain its responses, by submitting queries to a system being learnt. If a row constructed is the same as an existing row for another state B, one may conjuecture that the two states are the same; if not, one may attempt transitions from A and add more rows. The conjecture is tentative: an easy way to refute it is to take transitions from A and verify that states entered are the same which would be entered if the same transition from B is taken. If the two states prove different, a new column is added, distinguishing between them. After a while, no new rows/columns will be added and a learner has to ask a user to confirm that the learnt model is the expected one; a counterexample trace received in response starts another phase of learning.

Partial order, symmetry Expert input [9] is used to introduce an idea of partial orders on events, where a state entered by taking events in either order is the same or by writing temporal-logic constraints. A user may additionally introduce a *symmetry* abstraction: for a system with a number of phones it may not matter what those phones are, hence it is akin to an introduction of a universal quantification. This is similar to abstraction performed by [6] which checks for valid abstractions it by experiment.

K-tails(1) Reference [15] learns state machines by using a probabilistic K-tails learner. A non-probabilistic K-tails learner unifies a pair of states based on whether k states following from the considered pair are the same or not, where k has to be defined by a user. Probabilistic learner estimates the likelyhood that k-long tails are going to be the same; the result of learning is a Markov chain where probabilities on transitions reflect how likely a path is to be tra-

versed. It is suggested in [15] to remove edges which correspond to low probabilities under assumption that they reflect is atypical and potentially erroneous behaviour of a system.

K-tails(2) The K-tails algorithm (merging states followed by k-long sequences of events) has been modified in [16] to additionally merge states reached by transitions from the same state with the same outgoing label.

Markov learner Markov learner [16] learns Markov chains which include second-level probabilities, relating a pair of events to the probability of the event to follow.

GK-tails GK-tails algorithm [13] is a derivative of the k-tails one but supports conditions on transitions. This makes it possible for it to merge states if sequences of transitions from it have the same method calls but non-identical conditions, such as conditions in one tail implying conditions in the other one or one tail contained in another one.

Method of [24] Shimba [11] is a system permitting reverse-engineering software; a method dependence graph can be used by a user to select methods to collect traces from. Signatures of method calls and decisions taken are collected. Any object in a resulting message sequence chart can subsequently be converted into LTS using the method described in [24]. In this method, outgoing calls and condition checks are considered to be states; incoming calls and the outcome of condition checking are labels. States with the same labels are considered to be equivalent in the first instance. If the outcome is a non-deterministic LTS, [24] splits some of these states until a deterministic one is reached. In addition to LTS, [11] also generates high-level messages sequence diagrams with sub-diagrams and loops; this makes it possible for a user to inspect lengthy message sequence diagrams.

3.5 Feedback

At each stage in Figure 1 one learns more about a system under test and this information can be used to construct more tests or guide exploration. At a low level, [1, 2, 4, 5] explore concrete states; at a high level, one may use queries from an iterative learner to guide learning. This applies to [9] where one attempts to run queries constructed by [19] on a system under test.

Although [14] does not use an automated test generation procedure, when new tests are developed, an existing classifier can be applied to new traces; traces which are not classified (i.e. there is no cluster which matches branch decisions observed sufficiently closely), are submitted to a user for classification.

No papers perform feedback after abstraction of extracted traces before model building (denoted by *feedback(2)* in Figure 1); this can be explained by the fact that abstraction and model development tend to be tightly integrated.

3.6 Discussion

Given that sequences of method calls obtained from running a program are similar to message sequence charts, it is surprising how few of the papers surveyed in this paper use numerous existing techniques for building models from sample computations, such as [44, 24, 18] and those surveyed in [40, 29]. Most papers in this area focus on building models at the requirements construction phase, where sample computations are given by a user and a system attempts to build a model. Building models using dynamic analysis is different in that (1) dynamically obtained traces usually contain a lot more information and (2) one can run as many experiments with a system as one wishes, in contrast to requirements elicitation where a user is a human and one would like to limit the amount of information that person has to supply. The idea of active learning [14] proved to be effective in extracting models where only behaviours unknown to a classifier are submitted to a user; in the area of regular inference, [31] proposes an algorithm which aims to build a state machine while attempting to reduce the number of questions asked.

It would be interesting to compare models built by abstracting both states and transitions and those relying purely on transitions. A discrepancy could indicate that a part of an internal state is not explored in the learnt behaviour and help perform refactoring or add more tests.

Only papers learning algebraic object models and [13] introduce constraints on method call values; there is existing work which could be used to learn such constraints [39, 22].

Probabilistic learners are capable of ignoring 'anomalies' in the traces they are supplied with and still learn a reasonable automaton [16, 8]. One can argue whether this is always a good thing, since infrequent transitions may be infrequently traversed during testing and more likely to contain faults; identification of uncommon behaviour is a known research topic, [7] constructs tests violating axioms which are usually satisfied during testing.

4 Conclusions

A substantial number of papers have been published in the area of reverse-engineering software using dynamic analysis. The contribution of this paper is a summary of the techniques used in these papers, in the form of Table 1, which follows the data-flow diagram (Figure 1) describing the stages of the process of reverse-engineering of state-based models.

One of the outcomes of the comparison provided in this paper is that (1) few methods use existing work in the field of requirements engineering, where software models are built from sample interactions; (2) feedback can be very useful (as in Angluin learner) but is underutilised by many existing methods. Improving on these two aspects seems to be a promising direction to enhance the considered reverse-engineering methods.

Acknowledgements

This work has been sponsored by the EPSRC grant EP/C511883/1 "Automated abstraction of code into a state-based specification and test generation".

References

[1] H. Yuan and T. Xie. Automatic extraction of abstract-object-state machines based on branch coverage. In *1st International Workshop on Reverse Engineering To Requirements (RETR)*, 2005.

[2] T. Xie and D. Notkin. Automatic extraction of sliced object state machines for component interfaces. In *3rd Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, 2004.

[3] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *6th International Conference on Formal Engineering Methods (ICFEM)*, 2004.

[4] H. Yuan and T. Xie. Automatic extraction of abstract-object-state machines from unit-test executions. In *28th International Conference on Software Engineering (ICSE), Research Demonstration*, 2006.

[5] H. Yuan and T. Xie. Substra: A framework for automatic generation of integration tests. In *1st Workshop on Automation of Software Test (AST)*, 2006.

[6] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. *Lecture Notes in Computer Science*, 2743:431–456, 2003.

[7] T. Xie and D. Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005)*, pages 277–287, Nov. 2005.

[8] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In P. Frankl, editor, *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis (ISSTA-02)*, volume 27, 4 of *SOFTWARE ENGINEERING NOTES*, pages 221–231, New York, July 22–24 2002. ACM Press.

[9] B. Steffen and H. Hungar. Behavior-based model construction. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *VMCAI*, volume 2575 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2003.

[10] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *ICSM*, pages 155–164. IEEE Computer Society Press, 2005.

[11] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering java software systems. *Software—Practice and Experience*, 31(4):371–394, 2001.

[12] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 313–326, New York, May 19–25 2002. ACM Press.

[13] D. Lorenzoli, L. Mariani, and M. Pezzè. Inferring state-based behavior models. In *WODA '06 International workshop on Dynamic systems analysis*, pages 25–32, New York, NY, USA, 2006. ACM Press.

[14] J. Bowring, J. Rehg, and M. Harrold. Active learning for automatic classification of software behavior. In G. Avrunin and G. Rothermel, editors, *ISSTA*, pages 195–205. ACM, 2004.

[15] G. Ammons, R. Bodík, and J. Larus. Mining specifications. *ACM SIGPLAN Notices*, 37(1):4–16, Jan. 2002.

[16] J. Cook and A. Wolf. Automating process discovery through event-data analysis. In *Proceedings of the 17th International Conference on Software Engineering*, pages 73–82. ACM Press, Apr. 1995.

[17] J. Yang and D. Evans. Dynamically inferring temporal properties. In C. Flanagan and A. Zeller, editors, *PASTE*, pages 23–28, Washington, DC, USA, 7-8 June 2004. ACM.

[18] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 304–313. ACM Press, June 2000.

[19] D. Angulin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[20] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer Verlag, May 2001.

[21] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, FASE'05*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.

[22] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In L. Baresi and R. Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.

[23] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with blast. In M. Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 2–18. Springer Verlag, 2005.

[24] A. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, SE-2(3):141–153, Sept. 1976.

[25] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[26] N. Cataño and M. Huisman. Formal specification and static checking of Gemplus' electronic purse using ESC/Java. In L. Eriksson and P. Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 272–289. Springer-Verlag, 22-24 July 2002.

[27] F. Chen, N. Tillmann, and W. Schulte. Discovering specifications. Technical Report MSR-TR-2005-146, Microsoft Research, Oct. 2005.

[28] G. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in cilk programs that use locks. In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[29] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[30] R. Doong and P. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, Apr. 1994.

[31] P. Dupont, B. Lambeau, C. Damas, and A. van Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial intelligence, Special issue on Applications of Grammar Inference*, 2007. To appear.

[32] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[33] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and algirithms for the construction and analysis of systems (TACAS)*, Lecture Notes in Computer Science, pages 357–370. Springer Verlag, 2002.

[34] E. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Aspects of Computing*, 17(2):201–221, 2005.

[35] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R. Kutsche and H. Weber, editors, *FASE*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

[36] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Symposium on Principles of Programming Languages*, pages 17–32, 2002.

[37] T. Lau, P. Domingos, and D. Weld. Learning programs from traces using version space algebra. In J. Gennari, B. Porter, and Y. Gil, editors, *K-CAP*, pages 36–43. ACM, 2003.

[38] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of The IEEE*, volume 84, pages 1090–1123, Aug. 1996.

[39] K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In E. Najm, J. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450. Springer Verlag, 2006.

[40] K. Murphy. Passively learning finite automata, 23 Dec. 1996. `http://www.cs.berkeley.edu/~murphyk/Papers/FSMsurvey.ps.gz`.

[41] J. Nimmer and M. Ernst. Automatic generation of program specifications. In *International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy*, pages 232–242, 22–24 July 2002.

[42] D. Saff and M. Ernst. Mock object creation for test factoring. In C. Flanagan and A. Zeller, editors, *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 49–51, Washington, DC, USA, 7-8 June 2004. ACM, ACM.

[43] M. Taghdiri. Inferring specifications to detect errors in code. In *ASE*, pages 144–153. IEEE Computer Society, 2004.

[44] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 314–323. IEEE Computer Society Press, 2000.

[45] T. Win and M. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Laboratory for Computer Science, 25 May 2002.

[46] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *19th IEEE International Conference on Automated Software Engineering (ASE)*, 2004.

[47] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In N. Halbwachs and L. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05), Edinburgh*, volume 3440 of *LNCS*, pages 365–381. Springer-Verlag, Apr. 2005.