Inferring Finite-State Models with Temporal Constraints

Neil Walkinshaw, Kirill Bogdanov
Department of Computer Science, The University of Sheffield
E-mail: {n.walkinshaw, k.bogdanov}@dcs.shef.ac.uk

Abstract

Finite state machine-based abstractions of software behaviour are popular because they can be used as the basis for a wide range of (semi-) automated verification and validation techniques. These can however rarely be applied in practice, because the specifications are rarely kept up-to-date or even generated in the first place. Several techniques to reverse-engineer these specifications have been proposed, but they are rarely used in practice because their input requirements (i.e. the number of execution traces) are often very high if they are to produce an accurate result. An insufficient set of traces usually results in a state machine that is either too general, or incomplete. Temporal logic formulae can often be used to concisely express constraints on system behaviour that might otherwise require thousands of execution traces to identify. This paper describes an extension of an existing state machine inference technique that accounts for temporal logic formulae, and encourages the addition of new formulae as the inference process converges on a solution. The implementation of this process is openly available, and some preliminary results are provided.

1. Introduction

Abstract specifications that capture the behaviour of a software system are used for several important software development tasks. Developers can use them as a basis for communicating with each other, they are used for model-based testing, they can be inspected, and they can be used by model checkers to verify properties. If the model is complete and up-to-date, the aforementioned techniques can be used in harmony to ensure that the software ultimately behaves correctly.

In practice however a model is rarely available. Software development is often carried out under restrictive time constraints, which means that developers tend to concentrate on writing the source code, and do not have the time to generate models in tandem. Even if a model is generated, it is rarely kept up-to-date as the software evolves. As a result, validation, verification and maintenance are inevitably hampered; a lack of a model means that the tasks mentioned above need to be carried out at an implementation-level, and are therefore unlikely to highlight any high-level functional faults.

An automated approach to infer specifications of software behaviour would enable the application of these verification and validation techniques at a design level. Several inference techniques have been proposed, most of which rely on obtaining some set of execution traces that are in some way "representative" of the system behaviour in general [1, 5, 13, 20, 21]. One major problem that hampers the widespread application of these approaches is that, unless the system is trivial, it will often require a vast number of traces to produce an accurate result with any confidence. Particular properties of the underlying system that may be obvious to the developer can require a vast number of traces to infer in practice.

This paper describes a technique that substantially reduces the reliance upon large numbers of traces. It enables the provision of linear temporal logic (LTL) constraints that are known to hold for the target machine, alongside the usual sets of traces. These properties may be obvious to the developer, but require lots of traces to capture. The technique is an extension of a conventional state machine inference process, but uses a model checker to ensure that any intermediate hypothesis machines do not violate any of the supplied constraints. If they do, counter examples from the model checker are fed back into the inference engine, and the process is restarted.

The technique can either be run passively, where constraints and traces are provided a priori, or actively, where it will iteratively query the developer about the behaviour of the system as it "homes in" on an accurate result. The active approach forces the developer to consider scenarios that may not have been envisaged, and enables the addition of new LTL constraints and scenarios that either confirm or contradict the scenarios suggested by the inference technique. As such the active approach can also be used to address the acknowledged problem of identifying suitable temporal logic constraints for finite state verification tasks

[8]. The contributions are as follows: (1) a state machine inference process that incorporates LTL constraints; (2) an active version of the inference algorithm that allows for the addition of new LTL during the inference process and (3) an openly available proof of concept implementation that uses the SPIN model checker to check LTL constraints.

Section 2 presents an overview of the state of the art of reverse engineering behavioural models from software systems. Section 3 shows how LTL can be integrated into the process. Section 4 presents implementation details, and section 5 evaluates the practicality of the technique by illustrating it with respect to two case studies. Finally section 7 concludes the paper and discusses plans for future work.

2. Reverse Engineering Behaviour Models

This section provides an introduction to current reverse-engineering techniques. Due to space constraints it is not exhaustive, but it provides an introduction to the generic *state merging* technique, which forms the basis for most state machine inference techniques (as well as the one we present in this paper). It also includes a small running example to show how such techniques, which are based purely on execution traces, can be impractical. The example is of a simple text editor. In the text editor, once documents are loaded they can be edited. Documents can be saved only if they have been changed in some way. Other documents can only be loaded once the current document has been closed.

2.1. State merging

The majority of existing approaches to reverse engineer state machines adopt a *state merging* approach. These approaches, which are typified by Bierman's *k-tails* approach [3], work on the basis that two states (points in an execution trace) are equivalent and can be merged if their future behaviour is identical. This notion of equivalence is called the Myhill-Nerode relation [16]. The challenge of identifying a state machine from a set of traces is thus reduced to identifying pairs of points in the traces that may be suitable merging candidates. Each merge results in a machine that is more general, i.e. it represents a broader range of software behaviour.

To begin with, the set of traces is usually represented by an *augmented prefix tree acceptor (APTA)*. Usually the traces represent actual executions, but if the developer knows of particular traces that would be impossible, these can be added, and tagged as impossible. The APTA is constructed by representing the set of traces as a single tree, where each node in the tree represents a particular prefix. Figure 1 illustrates the APTA for a set of sequences that correspond to executions of a simple text editor. Valid traces belong to the set S^+, and invalid traces belong to S^-.

Initial traces

$$\begin{aligned} S^+ = \quad & \{< load, edit, edit, save, edit, exit >, \\ & < load, edit, save, close, load, exit > \\ & < load, edit, close, exit > \\ S^- = \quad & \{< close >, \\ & < load, edit, save, load >, \\ & < load, close, save >\} \end{aligned}$$

Augmented Prefix Tree Acceptor

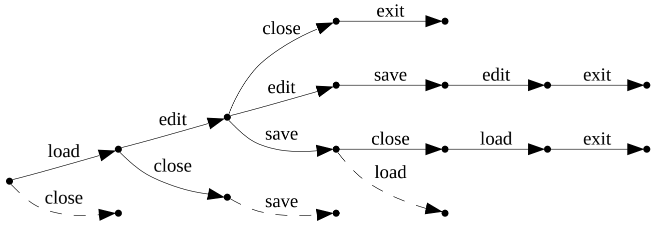


Figure 1. Augmented Prefix Tree Automaton

The state merging challenge is to select and merge the correct pairs of states that lead to an accurate state machine. Conventional (passive) merging techniques [3, 1, 5, 13] start from the APTA and produce a state machine in a single step. The accuracy of these machines depends on the number of relevant traces provided. Without enough traces to distinguish states that are different, these techniques can either fall prey to *overgeneralisation* (too many states that should be separate are merged together) or the APTA is so sparse that the resulting machine is very big because too many states have not been merged due to a lack of evidence of their equivalence.

In an effort to prevent this, active techniques [7, 20] attempt to garner the missing information from the developer and to validate state merges as they are encountered. For example, one could validate the merger of the two states with the outgoing transition "edit" by asking the question: "is the path $< load, save >$ a valid one? If the user says yes, this path (and other question-paths such as $< load, edit, edit, edit >$) are introduced to the new hypothesis machine that would not have been present in the previous version of the machine. This process continues until no new questions can be generated.

These active state merging techniques have been empirically shown to fare particularly well when the initial sample of traces is sparse [7, 20, 21] - which is often the case in dynamic analysis. Despite their increased accuracy, these active techniques can often also be impractical, because they can end up asking too many questions during inference. Large numbers of questions are particularly tedious when they have to be answered by a human (usually the developer).

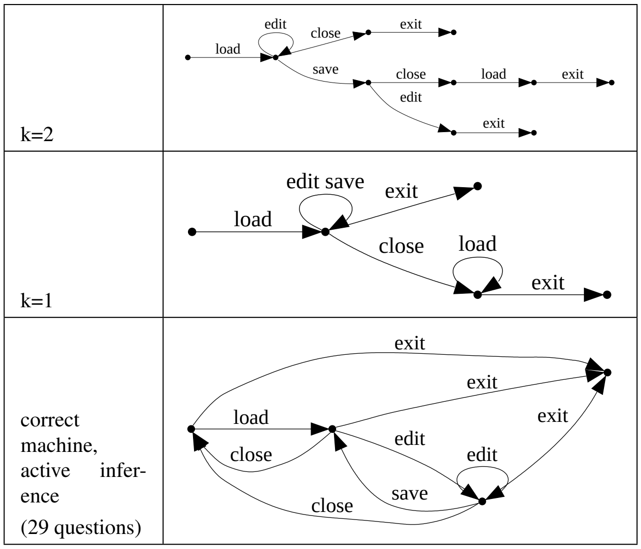


Figure 2. Inferred machines of text editor

2.2. Demanding input requirements

Current state machine inference approaches are unsatisfactory, because they are bound to produce inaccurate machines unless provided with an unrealistic amount of external knowledge about the target system. They are cumbersome because that information can only be supplied (a) in the form of program traces or (b) in the form of questions to some oracle (usually the developer). Dependence upon large numbers of relevant traces often renders them impractical.

Figure 2 contains the three inferred machines. The top two are generated by adopting the simple k-tails approach. The value of k is crucial. If it is too big, not enough states will be merged with each other, and the machine will not be general enough. This is demonstrated by the top machine. Conversely, if k is too low, the machine can be overgeneralised, as shown in the second machine (it permits saving without having edited the file).

The bottom machine is created by adopting Dupont *et al.*'s active QSM state merging approach [7] (using a slightly modified question generation algorithm [20]). Every time a pair of states are selected to be merged in the APTA (hyothesis machine), it generates a set of question sequences that would belong to the new merged machine. Answers from the user are added to sets S^+ (positive) and S^- (negative) as appropriate. If none of these conflict with the user's answers, the merge proceeds. Although the final machine is complete and correct, it requires a substantial 29 questions to gather the required information. Since this is a relatively simple machine, the number of questions will increase sharply as the target machine becomes more complex.

The restriction of traces and questions as the sole form of input for inference algorithms is often frustrating. Ensuring that the final machine adheres to a simple constraint can require large numbers of traces. As an example, to establish the simple property that a file can only be saved after it has been changed, positive examples are required to show that a single edit can occur before a save and that multiple edits can also occur before a save. Negative examples are also required to show that a save cannot happen after the file has been closed, and to show that two saves in a row cannot happen etc. Thus, the reliable inference of any non-trivial machine is often hampered by far too many traces or trivial questions.

3. Specifying and Checking LTL Constraints

This paper presents a constraint-based approach to reduce the input requirements of state merging techniques (both passive and active). The technique allows certain constraints on the target machine to be specified in linear temporal logic (LTL) axioms, and to be supplied alongside the conventional sets of valid and invalid traces. The conventional inference approach is augmented; every time a pair of states is merged, the resulting machine is checked with an LTL model checker to ensure that it does not violate any of the supplied properties. If it does, a counter example is supplied as a negative sequence to the learner. Section 3.1 provides a brief introduction to LTL, and section the use of model checkers to verify the conformance of a model to LTL properties.

3.1. LTL safety properties

Temporal properties are rules that govern the behaviour of a system in time. LTL [17] is a specification language that can be used to describe these properties. Besides the usual logic connectives $\wedge$, $\vee$, $\neg$ and $\rightarrow$, LTL also contains the following temporal operators:

- The *next* operator $\bigcirc$, where $\bigcirc\phi$ means that the property ϕ has to hold in the next state.
- The *global* operator $\Box$, where $\Box\phi$ means that the property ϕ has to hold for every future state.
- The *eventually* operator $\Diamond$, where $\Diamond\phi$ means that the property ϕ has to hold eventually.
- The *until* operator U, where $\varphi U \phi$ means that the property φ has to hold until property ϕ holds.
- The *release* operator R, where $\varphi R \phi$ means that the property ϕ is true until φ becomes true, or forever if φ is never true.

LTL properties can be divided into two categories [18]: those that must never be violated (safety properties) and those that must always hold (liveness properties). This dichotomy is useful because, depending on the type of property, different model checking approaches are required to ensure that a property holds. This work is concerned solely with the safety properties - we use LTL to specify properties that the inferred state machine must not violate.

Conventional state machine inference techniques often require many example execution sequences to suggest that a pair of states can or cannot be merged. As an example (from the text editor in the previous section), we consider the following requirement: *"a text file can only be saved once it has been edited"*. For the state-merging approach, it is very difficult to identify a set of valid and invalid traces that are required to produce a machine that satisfies the property. Although the set of traces provided in figure 1 all conform to the property, they fail to ensure that the final machine adheres to it (as shown by the machine where $k = 1$ in figure 2). The active approach does produce the correct machine, but only because it asks numerous questions which prevent the overgeneralisation. By adding additional sequences (such as $< load, save >$ is invalid, $< load, edit, save >$ is valid but $< load, edit, save, save >$ is invalid etc.), it eventually converges to a correct machine.

This information can however instead be expressed as a concise LTL expression, eliminating the challenge of identifying suitable sets of example traces. Instead of modelling a system in terms of state sequences (as is conventionally the case), we use LTL to model the system in terms in terms of sequences of events. This can be simply achieved by using a single variable e for the event, where each different event corresponds to a different variable value. Thus, the above constraint can be expressed as follows:

$(edit\, R\, \neg save) \wedge \Box(save \rightarrow \bigcirc(edit\, R\, \neg save))$

Other properties can be expressed in a similarly concise manner. Whereas, for example, 8 of the 29 sequences generated for the active inference merely confirmed that exit is always the final event, these could all be summarised by the constraint: $exit \rightarrow \Box(exit)$.

3.2. Identifying safety property violations

A number of model checkers exist that can establish whether a model conforms to a set of LTL properties [9, 15]. Model checkers ensure that a safety property is never violated by systematically searching through the states in the model, and checking each state against the set of safety properties. If they do detect a violation, they can provide a *counter example*, which is an execution path that leads to the violation.

For the sake of illustration, let the model inferred when k=1 in figure 2 be our current hypothesis, and the example properties shown in section 3.1 be our LTL specification. In this case, a merge has occurred that overgeneralises from the sample; it wrongly claims that, after a document has been loaded, *edit* and *save* can be used interchangeably until the document is closed or the program is terminated. A counter example is a path that is valid in the hypothesis, but invalid with respect to the specification - here $(edit\, R\, \neg save) \wedge \Box(save \rightarrow \bigcirc(edit\, R\, \neg save))$. The sequences $< load, save >$ and $< load, edit, save, save >$ from the hypothesis machine are both counter examples with respect to that property.

4. Constraint-based state machine inference

This section presents an algorithm that adapts existing state-merging algorithms to accept LTL constraints as well as conventional valid and invalid execution sequences. The presented approach not only ensures that the final machine conforms to given LTL properties, but uses counter examples from a model checker to extend the set of provided sequences as is necessary. The approach can be used with existing passive approaches that infer the machine in a single step, or it can be integrated into an active inference approach, that permits the addition of new properties during the inference process. The algorithm is presented in section 4.1. Section 4.2 contains details of a proof-of-concept implementation that works in conjunction with the SPIN model checker, and is openly available.

4.1. The inference algorithms

Conventional state merging algorithms (as described in section 2.1) start with the APTA (built from S^+ and S^-). They iteratively merge pairs of states until they end up with a minimal state machine where any further merges would produce a machine inconsistent with $S^+$and S^-. If the approach is passive, the pairs of states are selected in a single continuous process until the algorithm produces the final machine. If the approach is active, the hypothesis machine that is produced by each merge is used to formulate queries to the developer, who can provide feedback (usually in the form of scenarios) that (in)-validate the merge.

Figure 3 contains an overview of the inference process. It is based upon the conventional active state machine inference process [2, 7, 20], but is augmented by the steps la belled (1-4). The essential novelty of this is that, instead of relying on a human oracle for feedback during the inference process, we can use a model checker to provide feedback in the form of counter examples as well. As the inference process continues to refine the hypothesis machine, the human oracle can augment the set of LTL constraints as well.

Essentially, the algorithms that are presented here check the results of each merge to ensure that none of the LTL

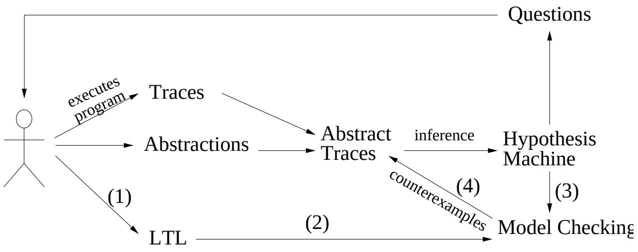


Figure 3. Inference process

properties are violated. If they are violated, S^- is augmented with a counter example and the algorithm recursively restarts. The active version provides the additional ability to manually augment the LTL set during the inference process. Thus, as the user is forced to contemplate new aspects of system behaviour that hadn't been considered when the initial LTL set was generated, new constraints and scenarios can be added to produce an accurate model, as well as a more complete set of accompanying LTL constraints. Figure 4 presents both the passive and active versions of the algorithm.

The passive *PassiveInfer* algorithm (see figure 4) starts off by generating an APTA from $S^+$and S^-. Every iteration, the *selectStatePairs* function selects a suitable pair of states to be merged (ensuring that the machine produced by merging the pair would not conflict with $S^+$and S^-). A number of established techniques, such as the Blue Fringe technique, exist to do this (see Lang *et al.* [11] and previous work by the authors [20] for pseudo code). The selected pair of states is merged using the *merge* function. This *merge* function differs from the conventional state-merging process (see Dupont *et al.* [7] for a more elaborate description). The process of determinising the (potentially non-deterministic) merged automaton is not equivalent to Hopcroft *et al.*'s standard conversion from a non deterministic machine to a deterministic one [10], because the merged machine is intended to describe a broader range of behaviour than the non deterministic machine it starts from.

So far, the process conforms to the standard state merging process. However, once the states have been merged, the *passiveInfer* algorithm differs. The hypothesis machine A' is checked by the *modelCheck* process to ensure that it conforms to the set of constraints provided by LTL. If any of the properties are violated by A', the model checker will return a counter example (or, depending on the model checker, a set of counter examples), that demonstrate this violation. If there are no counter examples, the process carries on merging pairs of states in A' until no further states can be found. If however counter examples *are* found by the model checker, they are added to the S^- set and the entire merging process is restarted. This carries on recursively un-

```
Algorithm PassiveInfer
  Input   $(S^+, S^-, LTL)$ sets of valid and invalid sequences, and a set of LTL properties
  Output  A minimal DFA $A$ consistent with an extended collection $(S^+, S^-, LTL)$
  Uses    $generateAPTA(S, S')$ Generates an augmented prefix tree acceptor from $S$ and $S'$
          $selectStatePairs(A)$ Selects merge candidates from $A$
          $merge(A, q, q')$Merges nodes $q$ and $q'$ in $A$, and ensures that the resulting machine is deterministic
          $modelCheck(A, LTL)$ checks model $A$ against $LTL$ and returns one or more counter examples
$A \leftarrow generateAPTA(S^+, S^-)$
while $(q, q') \leftarrow selectStatePairs(A)$ do
    $A' \leftarrow merge(A, q, q')$
    $Counter \leftarrow modelCheck(A', LTL)$
    if $(Counter \neq \emptyset)$
        $S^- \leftarrow S^- \cup Counter$
        return $PassiveInfer(S^+, S^-, LTL)$
    else
        $A \leftarrow A'$
return $A$
```

```
Algorithm ActiveInfer
  Input   $(S^+, S^-, LTL)$ sets of valid and invalid sequences, and a set of LTL properties
  Output  A minimal DFA $A$ consistent with $S^+$, $S^-$ and a potentially expanded set $LTL$
  Uses    all functions used by $PassiveInfer$ as well as:
          $modelCheck(query, LTL)$ checks that a single sequence conforms to LTL, returns a counter example if not
          $generateQueries(A, A')$ Returns a set of sequences over $A'$ that are classified differently by $A$
          $isValid(query)$ returns $\emptyset$ if $query$ is valid, otherwise returns the shortest negative part of $query$
          $newConstraintsFromUser()$ Returns new LTL constraints from the user, may be empty
$A \leftarrow generateAPTA(S^+, S^-)$
while $(q, q') \leftarrow selectStatePairs(A)$ do
    $A' \leftarrow merge(A, q, q')$
    $Counter \leftarrow modelCheck(A', LTL)$
    if $(Counter = \emptyset)$
        foreach $query \leftarrow generateQueries(A, A')$
            $userCounter \leftarrow isValid(query)$
            if$userCounter = \emptyset$
                $S^+ \leftarrow S^+ \cup query$
            else
                $LTL \leftarrow LTL \cup newConstraintsFromUser()$
                $S^- \leftarrow S^- \cup userCounter$
                return $ActiveInfer(S^+, S^-, LTL)$
    else
        $S^- \leftarrow S^- \cup Counter$
        return $ActiveInfer(S^+, S^-, LTL)$
    $A \leftarrow A'$
return $A$
```

Figure 4. Passive and active algorithms

til no further states can be merged, and no counter examples can be provided by the model checker.

The *activeInfer* algorithm (also in figure 4) is an active version of *passiveInfer*. As with *passiveInfer*, every time a pair of states is merged, it calls $modelCheck(A,LTL)$ to ensure that the merge does not contradict any of the current LTL clauses. At this point, with no counter examples from the model checker, *passiveInfer* would simply carry on merging. Instead, *ActiveInfer* proceeds to ensure that the merge is correct by querying the user. It generates a list of sequences by calling the $generateQueries(A,A')$ function. The question generation process is based on Dupont *et al.*'s QSM technique [6, 7] and produces scenarios that would be handled differently in A and A' (i.e. would be valid in one but not in the other). When faced with these sequences, the user can either confirm / reject the posed scenarios, or can provide an LTL property that accounts for a larger range of scenarios (including the posed scenario). Depending on how specific the supplied LTL properties are, this can substantially reduce the number of scenarios that need to be supplied and queried.

4.2. Implementation

The approach has been implemented as part of our StateChum state machine inference Java framework[1], that is openly available. A (currently very simple) interface is provided, that accepts a text file that specifies whether the inference should be active, an optional k-limit for the inference, as well as an initial set of valid and invalid execution sequences. It is assumed that the traces are encoded as sequences of functions at a suitable level of abstraction. This can be achieved by adopting the trace annotation process detailed by Ammons *et al.* ([1] - section 3).

LTL constraints are checked with the SPIN model checker [9]. Every iteration, the current hypothesis state machine is translated into PROMELA code (the modelling language for SPIN). Every time an LTL constraint is added, a *never claim* is generated that represents the Büchi automaton of the disjunction of all supplied constraints. SPIN checks the supplied model, and if it finds that the *never* claim is violated it returns a counter example. Although it is possible to carry on searching after a counter-example has been found, to identify further counter examples, our implementation currently restarts learning as soon as the first counter example has been identified.

It is important that the counter example supplied by the model checker is only invalid with respect to the final element (i.e. its prefix must be valid). The grammar inference process will presume that the prefix *is* valid, and if this is not the case it will produce a machine that accepts sequences that are invalid. In SPIN the shortest counter example is ensured by using the "breadth-first" switch to check for LTL properties [9].

4.3. Example

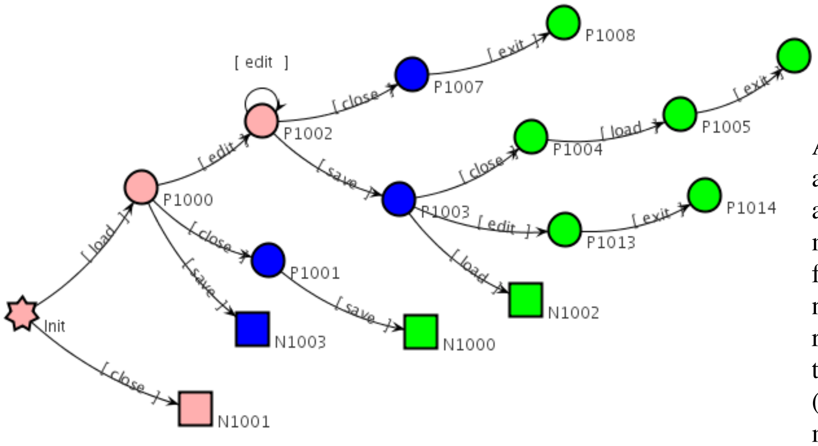
This example demonstrates the implementation, how LTL constraints reduce the number of questions, and how the questions encourage the addition of new and relevant LTL constraints. To do so we use the text-editor example introduced in sections 2 and 3. We start off with the traces $S^+$and S^- that are presented in figure 1 (although the negative sequences are useful, it is possible to start the inference from just positive sequences abstracted from program traces). We begin with the LTL constraint that was suggested in section 3.1: $(edit\, R \neg save) \wedge \Box(save \rightarrow \bigcirc(edit\, R \neg save))$.

Figure 5 provides a full listing of questions asked by the inference algorithm, answered by the user, and information provided by the model checker. The final result is the same as the final machine in figure 2. However, without the use of the model checker, the QSM technique posed 29 questions to the user, whereas here only 10 needed to be answered. The rest of the information about system behaviour was encapsulated in the LTL constraints that were provided by the user.

One of the barriers to the widespread adoption of logics such as LTL is the fact that it requires a substantial amount of effort to identify and denote those constraints that are relevant and useful [8]. This is lessened by the iterative nature of our active approach - by asking the developer questions they are forced to contemplate potentially unexpected system behaviour. For any questions that suggest an invalid system execution, there should exist a corresponding LTL expression that summarises why this is the case. As an example, the first question (step 3) asks whether one load can be succeeded by another. In practice the system can only load one file at a time. This property is expressed as the more general LTL rule (also shown at step 3). At step 9, the model checker provides a counter example for the current hypothesis model, because it breaks the rule that a file cannot be saved unless it has been edited since the previous save. However, the developer would notice that this constraint does not account for the fact that it is impossible to do anything to a file once it is closed, and expresses this at step 10. This rule eventually causes the model checker to suggest the counter-example at step 25. However, the developer would notice that, once the application has been terminated it is actually impossible to do anything, so this is encapsulated in the constraint at step 26. Besides an accurate state machine, the user also ends up with an accompanying set of relevant LTL constraints.

[1] http://statechum.sourceforge.net/

Step	Question to user	Manual input	Model checker
1			- load,save
2			- load,edit,save,save
3	load,load?	$\neg(\square(load \rightarrow \bigcirc(close\, R\, \neg load)))$	
4			- load,load
5	load,close,load,exit?	+ load,close,load,exit	
6	load,edit,exit?	+ load,edit,exit	
7			- load,edit,edit,save,save
8			- load,edit,save,load
9			- load,edit,edit,save,close,save
10		$\neg(\square(close \rightarrow \bigcirc(load\, R\, \neg(save \lor edit \lor close))))$	
11	load,edit,edit,save,close,load,exit?	+ load,edit,edit,save,close,load,exit	
12	load,edit,edit,close,exit?	+ load,edit,edit,close,exit	
13	load,edit,edit,exit?	+ load,edit,edit,exit	
14	load,edit,edit,edit?	+ load,edit,edit,edit	
15			- load,edit,close,save
16			- load,edit,save,close,save
17	load,edit,save,edit,exit?	+ load,edit,save,edit,exit	
18			- load,edit,edit,close,save
19			- load,edit,close,close
20			- load,edit,edit,close,close
21	exit?	+ exit	
22			- save
23	load,exit?	+ load,exit	
24			- exit, save
25			- load, close, exit, edit
26		$\neg(exit \rightarrow \square(exit))$	
27			- exit,edit
28			- exit,load
29			- load,exit,save
30			- load,edit,exit,save
31			- load,edit,edit,exit,save



A screen shot from iteration no. 12. The QSM approach employs the "Blue Fringe" merging approach [11], where the blue fringe of nodes moves down the PTA, and each node on the blue fringe is considered for merging with any of the nodes in its wake. The dark (blue) nodes represent the blue fringe nodes, the (red) nodes are those that have been passed over already. The (green) tail nodes are yet to be considered for merging.

Figure 5. Questions asked, along with information added by developer and model checker ('-' invalid sequence , '+' valid sequence), with screen shot from inference process

	No Model Checker			**Using SPIN**			
	Q	*Prec*	*Rec*	**Q**	**LTL**	*Prec*	*Rec*
JHotDraw	58	0.97	0.97	29	4	0.98	0.98
Jakarta CVS	137	0.86	0.97	44	20	0.97	0.7

Table 1. Results from JHotDraw and Jakarta CVS client

5. Evaluation and Discussion

Section 5.1 presents results from two well-known systems (JHotDraw and a Jakarta Commons CVS client), given a 'typical' set of executions, and a small set of LTL constraints to start with. This is followed by a discussion of the results, the merits of the technique, and areas that require further improvement.

5.1. Results

JHotDraw[2] is a widely used, open-sourced drawing framework that has been used in previous specification mining work by the authors [20]. The Jakarta Commons Net [3] is an open-source framework that implements a variety of commonly used network protocols. Lo *et al.* [13] used their implementation of a CVS client in the Jakarta framework to evaluate their state machine mining approach.

For this evaluation, the technique was supplied with a (realistically) sparse selection of (abstracted) traces from JHotDraw and the CVS system. The results show the difference between the presence and absence of the model checker. Although our implementation contains a number of question generation strategies that minimise the number of questions asked, we have opted for the most conservative, basic approach, to attempt to ensure that the question generation algorithm does not interfere with the results.

Due to space constraints we can only show the results summaries in this paper, but a more complete set of results (in the same format as figure 5) are available[4]. Table 1 shows the results, where "Q" refers to the number of questions. The results on the left were computed by applying Dupont's QSM technique [7] on abstracted event traces [20]. For the model checker-assisted approach, whenever a question was invalid, a concise LTL clause was supplied that summarised why (instead of the common approach of simply pointing out where the question becomes invalid). So, out of the 20 questions that were asked for JHotDraw using the model checker-assisted approach, only 4 were invalid, and required LTL, and out of the 44 CVS questions, 20 were invalid.

For both systems there was a substantial reduction in the number of questions asked of the user, which is one of the main aims of this technique. The reduction for JHotDraw was 50%, and for Jakarta CVS the reduction was 68%. By specifying only four LTL clauses, the number of questions asked was halved for JHotDraw. For the Jakarta CVS example, 20 small LTL clauses were specified, and the remaining 24 questions were answered directly by the developer.

The *Prec* and *Rec* columns show the precision and recall of the final machines. The notion of precision and recall is calculated by applying a test set from the (manually generated) target machine, and quantifying the proportion of sequences that are correctly accepted and rejected [12, 21, 22]. For JHotDraw, despite halving the number of questions, and only requiring four LTL clauses, the accuracy for the SPIN-assisted machine is slightly higher. For the CVS system, the SPIN-assisted results significantly improve on the precision results, but produce a lower recall result. In practice, this means the the final machine does not account for as many sequences as it should (this can happen when it fails to detect loops for example). However, the sequences that are accounted for in the final machine are more accurately accepted / rejected.

5.2. Discussion

Although generally positive, it should be clear that the results presented above are merely indicative of the performance of the technique in general. It is challenging to produce a controlled, systematic evaluation of this technique, because there are so many factors that contribute to its accuracy and scalability. From an inference perspective, the amount of information required by the model checker / developer depends upon the number of initial execution traces provided. Initial empirical results by the authors [21], along with work by Dupont *et al.* [6, 7] suggests that the QSM technique performs well with respect to a sparse initial set of traces, and the traces provided above were deliberately sparse. The amount of information that can be automatically supplied by the model checker depends upon how specific the supplied LTL constraints are, and when they are supplied during the inference process. If lots of constraints are supplied before the inference starts, a large number of questions can be avoided during the inference process. In our future work, we aim to produce a more comprehensive empirical evaluation.

One critique that can be levelled against the technique presented here is that it depends upon the ability of the developer to supply reasonable LTL constraints. It is often argued [8] that formal specification languages (such as LTL) have not been widely adopted because their notations are cumbersome to understand and apply in practice. In the context of the technique presented here, the difficulty of

[2] http://www.jhotdraw.org

[3] http://jakarta.apache.org/commons/net/

[4] http://www.dcs.shef.ac.uk/~nw/Files/asematerial.html

constructing LTL formulae is tempered by the facts that (a) the required LTL clauses usually follow a very simple template (e.g. the question is wrong because *A* cannot precede *B*) and (b) the questioning process provides a useful guide for the LTL writing process. It is usually much simpler and less tedious to compose a short LTL clause, than to answer a lot of questions that are negated by the same rule anyway.

During the collection of the above results it was observed that, very occasionally, counter examples can themselves contain invalid prefixes. If the hypothesis machine is incorrect in a number of areas, a counter example that highlights one area might traverse another part of the machine that is incorrect (but not captured in an LTL clause). As a result, when the counter example is fed back into the grammar inference algorithm, it contains an incorrect prefix that is presumed to be correct. For instance, let the hypothesis machine be the machine (k=1) in figure 2, and there is an LTL formula $\neg\Box(load \rightarrow \bigcirc(\neg load))$. A possible counter example $< load, save, save, close, load, load >$ starts with an erroneous sequence which is present in the hypothesis machine (two consecutive saves cannot happen in practice). The conventional learner would simply assume that the prefix $< load, save, save, close, load >$ is valid, and add it to S^+.

In the *PassiveInfer* algorithm this will not cause a conflict, and the result may still be more accurate than it would be without the faulty counter example, because an incorrect merge has been prevented. However, when using the $ActiveInfer$ algorithm, this can lead to problems later, during the inference process, the user provides a new LTL constraint / query answer that contradicts the previously added false prefix. The authors have addressed this problem by implementing the $ActiveInfer$ algorithm in such a way that counter examples are added to a special cache instead of S^- (in line 15 in figure 4), and that this cache is flushed every time the developer adds a new statement or manually answers a question. In this case, any potentially conflicting counter examples are removed, and the inference process can start afresh, working only from the reliable data supplied by the user.

6. Related Work

The main contribution of this paper is an approach to state machine inference from event traces as well as LTL constraints. A large number of techniques already exist to reverse engineer state machines from event traces, and many of these techniques have already been covered in section 2. To the best of the author's knowledge, there are no active reverse-engineering techniques that employ LTL and counter examples in the way that has been presented. There have however been a number of attempts to use other sorts of inputs to guide or constrain the inference process, and these are briefly presented here.

In the domain of grammar inference, Martins *et al.* [14] propose a "more powerful teacher". Instead of simply confirming or rejecting queries, they propose that the teacher should augment every negative answer with an expression that specifies an additional set of invalid sequences. These additional strings are specified by providing a sequence of elements (this can be a prefix, suffix, or general subsequence of the set of all invalid sequences). Although the rationale is the same, this approach still forces the user to consider the system in terms of its execution sequences, and to identify these invalid sequences manually. By adopting the LTL approach in this paper, the constraints do not have to be explicitly phrased as execution sequences, they can be expressed in more general terms, and negative examples of behaviour can be identified automatically by the model checker.

In their work on the QSM technique Dupont *et al.* [6, 7] suggest that the states in the state machine can be decorated with data constraints specified as LTL formulae. With these formulae, and a set of initial constraints, symbolic execution can be used to identify the path conditions that must hold for the rest of the states in the machine. These data constraints are then be used to prevent invalid merges. This work has inspired a lot of the work presented by this paper. The work presented here complements their work in two principal ways: (1) The approach uses a model checker, and counter examples are fed back into the inference process, and (2) this approach permits (encourages) the addition of new LTL constraints during the merging process to complement his state labelling approach.

Giannakopoulou *et al.* [4] have carried out a substantial amount of work on assume-guarantee reasoning, where model checkers are used as oracles to answer queries generated by a grammar inference learner. In their work, Angluin's inference algorithm [2] is applied to learn models (of assumptions about the environment of their system). Their approach and the one presented in this paper use the model checkers to answer the queries in different ways. Their approach uses a model checker to check that a given property holds for a component in the context of the learned assumption. We use the model checker to check that a set of given properties hold for the learned machine itself.

Uchitel *et al.* [19] describe a technique to synthesise models from scenarios and LTL properties. Instead of producing state machines, they produce *Modal Transition Systems*, which incorporate "maybe" transitions to account for transitions that are neither exercised by scenarios nor ruled out by safety properties. They suggest that these can be refined manually, however the process of exploring these models is performed on an ad-hoc basis (using inspection, animation and model-checking). Our approach presents a more systematic exploration approach, that aims to facilitate the process of identifying the missing information that is required to complete the model.

7. Conclusions and Future Work

Reverse engineering techniques will only be adopted in practice if they do not demand too much effort from the user. This paper has presented a technique that addresses this weakness in conventional state machine reverse-engineering techniques. It facilitates the inference of state machines by enabling the developer to add their LTL constraints during the inference process, and so reduce the amount of input required to recover a reasonably accurate machine. An accompanying tool has been implemented and demonstrated to be feasible for use with realistic systems.

Our work has so far concentrated on the use of safety-properties, to specify properties that should not happen in the target model. Our future work will explore the use of LTL constraints to synthesise positive traces. Thus, it would be possible to specify constraints that encapsulate valid traces, and these could be used to spur the question generation process.

Our inference framework has a large module devoted to the experimental analysis of inference algorithms. This has proved useful in past work that evaluates the accuracy of conventional algorithms [20, 21]. Our current focus is on developing a systematic, larger-scale quantitative study of the LTL-constrained inference technique in this paper. We intend to identify more precise measures of how much knowledge is embedded in the LTL constraints that are added (e.g. by looking at the complexity of the Büchi automata), and relating this to the accuracy of the final machines.

Acknowledgements The authors thank David Lo from the National University of Singapore for kindly providing the Jakarta CVS client system. This work is sponsored by EPSRC grant EP/C511883/1.

References

[1] G. Ammons, R. Bodík, and J. Larus. Mining specifications. In *POPL'02*, pages 4–16, Portland, Oregon, 2002.

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

[3] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21:592–597, 1972.

[4] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

[5] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3):215–249, 1998.

[6] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE TSE*, 31(12):1056–1073, 2005.

[7] P. Dupont, B. Lambeau, C. Damas, and A. van Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22:77–115, 2008.

[8] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[9] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[10] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[11] K. Lang, B. Pearlmutter, and R. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In *Proceedings of the International Colloquium on Grammar Inference (ICGI'98)*, volume 1433, pages 1–12, 1998.

[12] D. Lo and S. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'06)*, pages 51–60. IEEE Computer Society, 2006.

[13] D. Lo and S. Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, pages 265–275, 2006.

[14] A. L. Martins, H. S. Pinto, and A. L. Oliveira. Using a more powerful teacher to reduce the number of queries of the L* algorithm in practical applications. In *EPIA'05*, pages 325–336, 2005.

[15] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[16] A. Nerode. Linear automata transformations. *Proceedings of the American Mathematical Society*, 9:541–544, 1958.

[17] A. Pnueli. The Temporal Logics of Programs. In *18th IEEE Symposium on the Foundations of Computer Science*, 1977.

[18] P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994.

[19] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *ICSE*, pages 34–43. IEEE Computer Society, 2007.

[20] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *WCRE'07*, 2007.

[21] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Improving dynamic software analysis by applying grammar inference principles. *Journal of Software Maintenance and Evolution: Research and Practice*, 2008.

[22] N. Walkinshaw, K. Bogdanov, and K. Johnson. Evaluation and comparison of inferred regular grammars. In *ICGI'08*, 2008. (to appear).