Empir Software Eng
DOI 10.1007/s10664-012-9210-3

STAMINA: a competition to encourage the development and assessment of software model inference techniques

Neil Walkinshaw · Bernard Lambeau · Christophe Damas · Kirill Bogdanov · Pierre Dupont

Editor: Sandro Morasca

Abstract Models play a crucial role in the development and maintenance of software systems, but are often neglected during the development process due to the considerable manual effort required to produce them. In response to this problem, numerous techniques have been developed that seek to automate the model generation task with the aid of increasingly accurate algorithms from the domain of Machine Learning. From an empirical perspective, these are extremely challenging to compare; there are many factors that are difficult to control (e.g. the richness of the input and the complexity of subject systems), and numerous practical issues that are just as troublesome (e.g. tool availability). This paper describes the StaMinA (**Sta**te **M**achine **In**ference **A**pproaches) competiton, that was designed to address these

This work is supported by the EPSRC STAMINA project (EP/H002456/1), the EPSRC REGI project (EP/F065825/1), the Regional Government of Wallonia (GISELE project, RW Conv. 616425) and the MoVES project (PAI program of the Belgian government).

N. Walkinshaw (✉)
Department of Computer Science, The University of Leicester, Leicester, UK
e-mail: nw91@leicester.ac.uk

B. Lambeau · C. Damas · P. Dupont
ICTEAM Institute, Université catholique de Louvain (UCL), Louvain-la-Neuve, Belgium

B. Lambeau
e-mail: bernard.lambeau@uclouvain.be

C. Damas
e-mail: christophe.damas@uclouvain.be

P. Dupont
e-mail: pierre.dupont@uclouvain.be

K. Bogdanov
Department of Computer Science, The University of Sheffield, Sheffield, UK
e-mail: k.bogdanov@dcs.shef.ac.uk

Published online: 25 May 2012

problems. The competition attracted numerous submissions, many of which were improved or adapted versions of techniques that had not been subjected to extensive empirical evaluations, and had not been evaluated with respect to their ability to infer models of software systems. This paper shows how many of these techniques substantially improve on the state of the art, providing insights into some of the factors that could underpin the success of the best techniques. In a more general sense it demonstrates the potential for competitions to act as a useful basis for empirical software engineering by (a) spurring the development of new techniques and (b) facilitating their comparative evaluation to an extent that would usually be prohibitively challenging without the active participation of the developers.

Keywords Model inference · Software specification · Competition · State machines

1 Introduction

Models are crucial for the effective development and maintenance of software systems. They describe the system requirements at a level of abstraction that is understandable to the developers, providing them with a definitive point of reference. Models of software behaviour, which are the subject of this paper, are particularly valuable, because they can form the basis for powerful automated techniques for tasks such as verification, validation and refinement (Lee and Yannakakis 1996; van Lamsweerde 2009).

Despite their apparent advantages, models are often neglected during the software development process. Generating an accurate model for any non-trivial system can be prohibitively time-consuming. Given the routine time and cost pressures involved in software development, resources are usually invested directly into the rapid development and implementation instead. The major downside in doing so is that developers are consequently forced to understand the system in terms of its source code, and to resort to ad-hoc verification and validation techniques that are invariably more time-consuming and less effective than their automated counterparts.

Numerous techniques have been developed that attempt to address this problem by reducing or even eliminating the human effort involved in producing useful models. These can be used in either a forward- or a reverse-engineering context, using a selection of "examples" of software behaviour (either in the form of hand-supplied scenarios, or recorded program traces) to infer models with the help of algorithms from the domain of Machine Learning. Although the idea dates back to the work by Biermann and Feldman in the mid-seventies (Biermann and Feldman 1972; Biermann and Krishnaswamy 1976), research in the area has only recently increased in intensity, in large part due to advances in the Machine Learning domain that have made it possible to infer larger models to a greater degree of accuracy. State-based software model inference techniques have taken a number of directions. Many of these are substantial adaptations of Biermann and Feldman's original 'k-tails' algorithm (Ammons et al. 2002; Cook and Wolf 1998; Lorenzoli et al. 2008; Reiss and Renieris 2001). Some approaches have focussed on adapting alternative Machine Learning algorithms including probabilistic inference approaches to learn Markov models or PFSAs (Cook and Wolf 1998; Lo and Khoo 2006). Others have capitalised on recent advances in a subfield of Machine Learning known as Regular Grammar

Inference (Damas et al. 2005; Dupont et al. 2008; Lambeau et al. 2008; Raffelt et al. 2009; Shahbaz and Groz 2009; Walkinshaw and Bogdanov 2008; Walkinshaw et al. 2007). The above list of techniques is merely a small sample of the plethora of techniques to have emerged.

The emergence of so many techniques gives rise to the question of which one is best-suited for the task. Providing an empirically sound answer to this question is difficult. There are numerous factors that may affect the performance of a given technique, such as the "richness" of the input, or the complexity of the software system in question. Then there are also numerous significant practical factors. Implementations that are associated with a given technique may not be openly available. They are often complex to implement, or might have parameter settings that require a lot of algorithm-specific knowledge to use effectively. As a consequence, there is no convincing empirical basis by which to choose model inference techniques, and to guide further research in the area.

This paper describes the StaMInA (State Machine Inference Approaches) competition, which is intended to provide an empirically sound basis for the comparison of techniques for the inference of models in the form of Deterministic Finite Automata. The rationale for adding a competitive element (as opposed to simply establishing a standard benchmark) was to (a) increase participation and (b) encourage participants to develop technological advances that would result in techniques that would advance the state of the art. The competition was a success - it attracted numerous competitors, and the winning technique made significant advances on the other competitors and on the baseline. The winning technique is described in detail in an accompanying paper in this issue.

This paper provides an overview of the competition, and shows how competitions can be useful drivers for the empirical comparison of software-engineering techniques. Section 2 introduces finite state machines – the type of model that is produced by the techniques compared in the competition. It also provides a brief overview of previous attempts and competitions in the domain of Machine Learning that have sought to compare different techniques. Section 3 provides an overview of the StaMInA competition framework, including a description of the protocol, the baseline technique, and the web infrastructure. Section 4 provides an analysis of the submissions to the competition and a comparison of the techniques (the winning technique is described in detail by its authors in an accompanying paper). Section 5 provides a discussion of the competition, including threats to validity and providing some suggestions for future competitions. Section 6 presents some related work. Finally, Section 7 provides conclusions and discusses future work.

2 Background

This section provides an introduction to the problem of state machine inference, and discusses the characteristics of software models that make them especially difficult to infer. It presents a brief overview of the Blue-Fringe (also known as RedBlue) algorithm (Lang et al. 1998), which has already been extensively used for software model inference (Damas et al. 2005; Dupont et al. 2008; Lambeau et al. 2008; Walkinshaw and Bogdanov 2008; Walkinshaw et al. 2007) and forms the baseline for this competition.

2.1 Deterministic Finite Automata and Their Languages

Software behaviour is commonly modelled in sequential terms, i.e. the sequences of permissible and impermissible events or inputs/outputs that constitute its functionality. These sequences are usually modelled with the help of Deterministic Finite Automata (DFA).

Definition 1 A Deterministic Finite Automaton is a quintuple $(Q, \Sigma, \Delta, F, q_0)$, where Q is a finite set of states, Σ is a finite alphabet, $\Delta : Q \times \Sigma \rightarrow Q$ is a partial function and $q_0 \in Q$. F is a subset of Q and represents the set of final (accepting) states where the system may terminate. A DFA can be visualised as a directed graph, where states are the nodes, and transitions are the edges between them, labelled by their respective alphabet elements.

In practice, software systems are often modelled with a closely related formalism: The Labelled Transition System (LTS) (Keller 1976; Magee and Kramer 1999). This is effectively a DFA where $F = Q$. These can be used to model a subset of the languages that can be modelled by DFAs, where every prefix of an accepting sequence is also accepted (this is known as the set of *prefix-closed* languages). Throughout the rest of this paper, the term *state machine* refers to both DFAs and LTSs.

When discussing the *behaviour* of a DFA, we are referring to the possible (and impossible) sequences of elements in Σ (denoted Σ^*). The set of all possible sequences in a DFA is referred to as its *language*.

To formally define the language of a DFA, we draw on the inductive definition for an extended transition function $\hat{\delta}$ used by Hopcroft et al. (2007). For a state q and a string w, the extended transition function $\hat{\delta}$ returns the state p that is reached when starting in state q and processing sequence w. For the base case $\hat{\delta}(q, \epsilon) = q$. For the inductive case, let w be of the form xa, where a is the last element, and x is the prefix. Then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.

Definition 2 The language $L(A)$ of a DFA A is the set of strings reaching an accepting state from its initial state. Given the extended transition function, $L(A)$ is defined as follows: $L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$. The complement of a language L is denoted L^C (i.e. the set $\Sigma^* \setminus L$ of strings that do not belong to L). Sequences $w \in \Sigma^*$ for which $\hat{\delta}(q_0, w)$ is not defined are considered to be rejected by the automaton.

2.2 The Inference Challenge

Despite their benefits, state machines are rarely produced during development due to the amount of effort involved. Instead, developers will often only document the system in a selective, less formal manner. The absence of sufficiently comprehensive documentation means that they are forced to resort to ad-hoc techniques for verification and testing tasks.

The desire to automate or at least partially automate the generation of state machines has driven the development of numerous automated model inference approaches (Ammons et al. 2002; Cook and Wolf 1998; Damas et al. 2005; Dupont et al. 2008; Lambeau et al. 2008; Lo and Khoo 2006; Lorenzoli et al. 2008; Reiss and Renieris 2001; Walkinshaw and Bogdanov 2008; Walkinshaw et al. 2007). These

approaches operate by observing examples of the software behaviour, either supplied by the developer as scenarios, or by recording actual program traces. The challenge is to, given such a set of examples, produce a model that has a similar language to the idealised (hidden) model of the system.

To present this more formally, we denote the (hidden) target DFA as A. The set of examples is presented in the form of a set of sequences in Σ^*. In the software-engineering domain these may be program traces, or scenarios that have been supplied by the developer. A set of examples can be divided into two sets of sequences; one set S^+ is the set of examples that is known to belong to $L(A)$, and the other set S^- is known to belong to $L(A)^C$ (these represent valid and invalid sequences respectively).

To guarantee the inference of an accurate model, the contents of S^+ and S^- need to satisfy certain properties. In their early work on regular grammar inference (regular grammars are DFAs), Oncina and Garcia showed that their Regular Positive Negative Inference (RPNI) algorithm (Oncina and Garcia 1992) would be guaranteed to produce an accurate result if the set $S^+ \cup S^-$ is structurally complete and *characteristic* of the target model (Dupont et al. 1994). In other words, they would need to contain a sufficiently diverse set of samples for (a) samples in S^+ to reach every state transition and accepting state, and (b) samples in S^- to distinguish every pair of non-equivalent states.

This is the root of the inference problem; in most practical situations the example sets are highly unlikely to contain a sufficiently diverse range of examples to fulfil the above criteria. The states in a DFA model of a software system can require a highly specific choice of sequences to distinguish between each other, which is unlikely to be present in a set of examples unless it contains some detailed prior knowledge of the system (Walkinshaw et al. 2008). Given the inevitable incompleteness of the set of examples, the challenge is to infer a model that is at nonetheless accurate.

The notion of what constitutes an 'accurate' model is not absolute. The ability to obtain models that are guaranteed to be *exact* has been shown to be intractable (Angluin 1978; Gold 1978). However, early empirical work by Lang (1992) showed that it *is* possible to infer approximate models. Thus, the goal for practical model inference techniques is not to obtain exact models, but merely to induce good approximate models.

This is what forms the basis for the StaMInA competition. By allowing for a small degree of error, the challenge is to identify those techniques that are best able to infer *reasonably* accurate DFAs for given sets of examples.

2.3 The State-merging Approach

One family of techniques that has proven to be relatively successful for inferring models from examples is the *state-merging* approach. The basic idea is to lay out all of the examples in $S^+ \cup S^-$ into a *prefix-tree acceptor* (PTA), a tree-shaped automaton where any two sequences with the same prefixes share the same root. The PTA is a state machine that exactly represents the given set of sequences. The challenge is to produce a more general DFA that can not only correctly accept or reject strings in $S^+ \cup S^-$, but also correctly classify further unseen examples. This is achieved by selecting pairs of states that are deemed to be equivalent, and merging them. In this context, the task is ultimately to find those groups of states in the PTA that are

equivalent; the search for an accurate DFA can thus be characterised as the search for a suitable partition of the set of states in the PTA, where equivalent states form a single block in the partition (Dupont et al. 1994).

This is the basic process that is adopted by the Blue-Fringe technique (Lang et al. 1998) (a specific variant of what is often referred to as the EDSM algorithm), which is used as a base-line technique for this competition and was (before the competition) considered to be the state of the art in model inference. Since it is practically infeasible to compare every pair of states, successful techniques rely on adopting an effective strategy to select those pairs of states that are most likely to be equivalent. The strategy employed by the Blue-Fringe technique (see Appendix A for details) is to confine the comparison of state-pairs to a small subset of states close to the root of the PTA. Pairs of states are selected to be merged following the heuristic that those with the greatest overlapping suffixes are most likely to be equivalent. As will be shown in the results of this competition in Section 4, most of the successful competing techniques (including the winning one) are state-merging variants, and significantly improve on the performance of the Blue-Fringe algorithm by adopting different strategies to select state pairs.

2.4 The Evaluation and Comparison of Inference Techniques

Empirically evaluating the performance of model inference techniques is challenging, because their performance can be affected by numerous factors. The size and complexity of the target machine (the number of states and transitions, and the size of Σ) are indicators of this complexity, as is the depth (the longest "shortest walk" from the initial state q_0 to any other state). Besides the target model characteristics, the "richness" of the training set (a set of sequences and class labels indicating whether or not they belong to the language of the target) is another key factor; the more information about the target that it contains, the easier it is to infer the states and transitions in the final model. This explains the difficulty in reliably assessing the performance of a technique. All of these variables need to be controlled, to ensure that the experiment is not unfairly biased (either for the better or worse).

Within the domain of software engineering, model inference techniques are usually evaluated on a case-study basis (e.g., Lo and Khoo (2006) used a CVS system, Walkinshaw et al. (2007) used a drawing tool, and Damas et al. (2005) used a water pump controller). A real software specification is selected as the target for the inference technique, and is used as the basis for eliciting a set of traces/scenarios. Although this serves the purpose of establishing the feasibility of the technique, it fails to provide any reliable basis for comparison with other techniques.

In the machine learning domain, this problem has been addressed by organising competitions. Competitions offer a means to compare different techniques with respect to the same set of target models, with the same set of samples. Most competitions follow the example set by the successful Abbadingo-One competition (Lang et al. 1998) held in 1997. The organisers generate a set of quasi-random target models, but the competitors are only given a training set, and a test set (a set of sequences without labels). Once the competitors have induced a model with the training set, they classify each string in the test set as either belonging or not belonging to their hypothesis model. This results in a binary string, which they submit to the competition web server. If their classifications are at least 99% correct, they are deemed to have successfully inferred the model.

The target models and their respective training/test sets are usually arranged into categories that correspond to different levels of difficulty. The techniques to correctly infer the most difficult models are selected as winners. The Abbadingo-One competition was won by the Blue-Fringe algorithm by Price, which, before StaMInA, was deemed to be the current benchmark for grammatical inference algorithms (the algorithm is described in Appendix A).

Unfortunately, competitions such as Abbadingo-One have been unsuitable for the evaluation of inference techniques that are intended to infer models of software systems. There are two underlying reasons for this: (1) the target models do not exhibit characteristics that tend to arise with software models, and (2) the method used to generate the training and test sequences can lead to accuracy scores that are misleading. These two issues are elaborated below.

Model Generation The alphabets used to generate the random machines in Abbadingo-One are only binary ($\Sigma = \{a, b\}$), meaning that each state can have at most two outgoing edges. The complexity of the target machines is modulated by a nominal value n. Given this value, a random degree-2 directed graph (representing the transition structure of the machine) is generated on $\frac{5}{4}n$ nodes. The edges between states are inserted at random, and then a sub-machine consisting only of the useful states (states that are reachable, and themselves lead to accepting states) and the transitions between them is selected. For a given n, this results in a set of machines with roughly the same depth and size.

These machines are however not at all representative of typical software models. DFAs that model software systems involve state transitions that may be triggered by any of a large number of events (mouse-clicks, function names, IO events, etc.). The size of the alphabet, and the number of outgoing transitions from a given state can be very large, and vary significantly from state to state. The previously described random graph generation algorithm produces a homogeneous network, where any pair of states are equally likely to be connected. In practice, software DFAs do not obey this law; certain states represent *"hub"* states, e.g. the root of a menu system, or a defacto error-handling state, whereas other states may be intermediate parts of a longer sequence of specific operations.

Training/Test Sample Generation Generating random training and testing samples for past competitions has been straightforward. The approach for competitions such as Abbadingo has been to simply generate random sequences without regard to the target model (it is only used to assign labels to the random sequences). Essentially, this has involved producing random binary sequences (because the alphabet only contains two elements) up to a prescribed length, and to subsequently use the target model to classify them as "accept" or "reject". The nature of the synthesised models is such that the resulting sets of sequences are roughly balanced between examples that lead to an accept state and examples that do not.

If, however, we are using state machines that are more representative of software models, this balance would no longer hold. This is down to the factors that are characteristic of software models—a large alphabet and an uneven distribution of transitions. As the alphabet size increases, the set of possible sequences up to a given length in Σ^* that could be chosen as possible examples grows exponentially. At the same time, the fact that the set of alphabet elements that are possible from a given state is often a highly specific subset (and that this will vary substantially from state

to state) means that only a highly specific subset of sequences in Σ^* are liable to lead to an accept state. Ultimately, given such state machines, the current standard sample generation approach would result in sets of sequences where a substantial majority of the sequences would lead to reject states. This would result in an equally unbalanced assessment of the state machine accuracy (for example, a DFA that trivially rejects everything would score very well with a test set generated in this manner (Walkinshaw et al. 2008)).

3 The StaMInA Competition

The aim of the StaMInA competition is to identify the best inference technique for models of software systems. Specifically, the aim is to identify the technique that is best able to infer accurate DFAs with reasonably large alphabets, and is able to infer accurate models from a relatively sparse set of examples. *From an empirical software engineering standpoint, the competition can be seen as an experiment, where the competing techniques are the subjects, and the size of the alphabet and sparsity of the examples are the control variables.*

The competition is presented to the competitors in the form of a 20 problem-sets of varying difficulty, where each set contains five hidden randomly generated DFAs that are about 50 states in size. For each DFA, competitors are provided with a training set. Having used the sample to infer a hypothesis model, competitors are then provided with a test set, and each test sequence is then classified as accepted or rejected by the hypothesised model. The solution is then submitted to the competition webserver as a binary string, where a '1' represents a test that is accepted by the hypothesis, and a '0' represents a test that is not accepted. This is compared to the correct binary string to establish whether the hypothesis is a sufficiently accurate approximation. The aim is to solve the hardest problem set (also referred to as a 'cell' in in Table 1), by accurately inferring all five models (the process of establishing this accuracy is also discussed below).

Participants can register one or more inference algorithms to attempt on these problem sets. The performance of each algorithm is evaluated independently. The risk of using one algorithm to tune another is attenuated by the limited amount of feedback (in the form of a bit string)—as discussed in Section 5.1.

3.1 Synthesis of Target Models

The task of generating a random DFA that shares the characteristics of a software system is not trivial. As mentioned previously, conventional algorithms (c.f. those

Table 1 Table with mean BCR results for the five challenges in each cell, as computed by the baseline Blue-Fringe technique

$\lvert\Sigma\rvert$	Sparsity			
	100%	50%	25%	12.5%
2	0.99 (1)	0.95 (1)	0.67 (3)	0.66 (3)
5	0.97 (1)	0.78 (2)	0.59 (4)	0.52 (4)
10	0.93 (1)	0.64 (3)	0.51 (4)	0.5 (4)
20	0.91 (1)	0.63 (3)	0.54 (4)	0.51 (4)
50	0.81 (2)	0.64 (3)	0.57 (4)	0.5 (4)

Scores are associated with a difficulty grade from 1 (easiest) to 4 (hardest)

used in grammatical inference competitions (Lang et al. 1998)) generate transition structures that are homogeneous; there is a uniform probability that any pair of states is connected by a transition. In reality this is unlikely to be the case—in software systems certain states invariably play a more central role than others.

For this competition we have developed a customised algorithm to generate random DFAs that share characteristics that are typical for software models. To gain an insight into the key characteristic of software state machines, a small informal survey of existing state machine models was carried out. Its findings are discussed below. This is followed by a description of the random DFA generation algorithm itself.

3.1.1 Observed Software Model Characteristics

To gain an insight into the characteristics of a "typical" software model, a diverse sample of 20 software models was analysed, sourced primarily from publications involving state machines and RFCs (memoranda that are generally used to specify standard network protocols such as SSH and FTP). The rationale for this was to obtain an idea of the relationship between particular factors such as alphabet size and depth, so that these could be used to parameterise the DFA generation algorithm.

The DFAs were analysed by encoding their transition structures in such a way that they could be analysed by the iGraph extension for the R statistics framework.[1] The main purpose of the results is to act as guidelines for the generation algorithm; any statistically justified statistics about general software models would require a much larger set of models. The subject DFAs that formed the basis for this precursory analysis, along with an informal report on its results have been made available.[2] The main conclusions are briefly listed below.

1. **Alphabet size.** There was no statistically significant relationship between the size of an alphabet and structural features, such as the number of states/transitions, or the depth of the DFA.
2. **Relationship between depth and states.** There was a strong relationship between the depth (the longest shortest path from q_0) and number of states, modelled as follows (as established by linear regression): $depth = (0.36 * states) + 1.3$ ($R^2 = 0.75$).
3. **In/Out degrees for states.** No statistically significant relationship was found between the in- and out-degrees for states. However, a visual inspection of the transition structures clearly indicated that some states tend to play a more central role than others. This was confirmed by plotting histograms of Kleinberg's "hub" and "authority" scores (Kleinberg 1999) for each state. These showed that although most states play a relatively passive role in the state machine—perhaps being an interim state as part of a sequential process with one or two in/out-going transitions, there are consistently one or two states that have a high hub or authority score. These states tend to represent the starting or finishing points for a range of possible sequences of events.

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

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

3.1.2 Model Synthesis

The above observations were used as guidelines to develop an algorithm to synthesise random DFAs that have similar characteristics to those found in realistic software systems. In other words, the models have to have a reasonably large alphabet, with some states and transitions that are more 'central' than others. This is on top of the conventional constraints on DFAs (they should contain no equivalent states, they should be deterministic, and every state must be reachable from the initial state).

Generating graphs with uneven edge distributions is not trivial, and has been the subject of much study in social network research, where certain nodes (often representing web pages, or humans in a social environment) will often feature more centrally than others. Following this observation, the organisers developed a random DFA generator that builds on an existing algorithm to synthesise social networks (the Forest-Fire algorithm (Leskovec et al. 2007)).

In broad terms, the original Forest-Fire algorithm produces graphs with uneven edge distributions by iteratively selecting a random "ambassador" node, adding a new node, and subsequently adding edges to a selection of nodes that are linked (or transitively linked) to the ambassador node. As nodes are added, this produces clusterings of edges around certain nodes. Our extension takes these unlabelled graphs, adds transition labels, and ensures that the final model is deterministic, does not contain equivalent states, and that all states are reachable. States are randomly labelled as final as they are added. Further details are provided in Appendix B.

Although the synthesised models adhere to the characteristics observed in Section 3.1.1, the claim that they are representative of software models in general is difficult to validate. This must be taken into account when interpreting the results of the competition, and is discussed more fully in the 'threats to validity' section (Section 5.1).

3.2 Generation of Training and Test Sets

As stated in Section 2.4, the algorithms that are used to generate random samples for previous competitions are no longer suitable for DFAs with large alphabets and varying out-degrees, because they fail to generate a sufficiently large number of valid sequences. The sampling procedure used in this competition computes direct walks over the target machine as opposed to randomly combining elements of the alphabet. The procedure can be summarized as follows:

1. Using a random walk algorithm (see details below) a first sample is generated from the target DFA. The sample contains exactly 20,000 sequences but may contain duplicates. This is nonetheless established to be a sufficiently populous pool to enable the baseline Blue-Fringe algorithm to infer an accurate models.
2. The sequences are equally partitioned in two disjoint sets (satisfying a positive/negative balance). One is designated as the training pool, and the other as the test pool. Any duplicate sequences are removed from the test pool to avoid influencing the performance metric.
3. The final testing sample is computed by randomly selecting 1,500 sequences from the testing pool. This number was chosen by experimenting with the test results produced with respect to the baseline Blue-Fringe algorithm; larger numbers of sequences were found to have no effect on the reported test accuracy.

4. The training sample generation is parameterised by a desired *sparsity* value. As was explained in Section 2, the difficulty of an inference problem is determined not only by the characteristics of the target machine, but also by the completeness of the given training set. In our competition, we incorporate four levels of sparsity: 100% is a notionally rich sample, and 50%, 25%, 12.5% are subsets. The calibration of the number of training samples required for 100% is discussed in Section 3.3.2.

3.2.1 Random Walk Algorithm

A random walk algorithm has been implemented to generate the sample pool described above. It generates positive sequences by walking the automaton from the initial state, randomly selecting outgoing state transitions from a uniform distribution. The set of sequences is generated in such a way that the distribution of sequence lengths is approximately centered on $5 + depth(automaton)$, ensuring a high probability of generating samples that fully cover the states and transitions of the model in question (this is inspired by the Abbadingo competition (Lang et al. 1998)). This is achieved by terminating a walk with a probability of $1.0/(1 + 2 * outdegree(v))$ every time some terminal state v is reached (this probability was chosen by experimentation).

Negative sequences are generated by editing positive strings obtained as above. Three kinds of edits are used: substituting, inserting, or deleting a symbol. In all cases, the edit-location is chosen from a uniform distribution on the sequence length. The number of edits is chosen with a Poisson distribution centred on three, and the edit kind with a uniform distribution. The sequence is simply discarded if the edited version still ends in an accepting state.

3.3 Competition Setup and Calibration

The two key factors that can affect the performance of a model inference algorithm are (1) the complexity of the target model and (2) the richness of the sample set. To gain insights into the respective strengths of the competing techniques, these two factors have to be modulated carefully. This is especially the case with DFA complexity, which is a compound property that is affected by several properties of the DFA. The rest of this subsection describes how these factors were regulated for the competition.

There are many ways in which one can define the term ‘complexity’ with respect to DFAs, such as the number of transitions, the manner in which the transitions are arranged in the DFA, the number of states, and the alphabet size. By using our adaptation of the Forest Fire algorithm, all of these variables apart from the number of states and the size of the alphabet are implicitly factored into the model generation process. The number of states and alphabet size can be controlled explicitly. The decision was taken by the organisers to modulate the complexity of the DFAs only in terms of their alphabet size, and to keep the number of states at approximately 50 states. In generating the models, a slight variance of ± 9 was allowed and the true size was not disclosed to competitors (to prevent them from tailoring their technique to focus exclusively on models with exactly 50 states).

The two core problem factors, alphabet size and sample sparsity, are controlled to produce 20 problem categories of varying difficulty. The size of the alphabet is

adjusted to either 2, 5, 10, 20 or 50. The sparsity of the sample ranges from 100% to 50%, 25% and 12.5%. For each combination of alphabet size and sample sparsity, five random DFAs are generated, resulting in 100 DFAs in total. The categories are shown in the column/row labels in Table 1.

Having generated the set of target models, and arranged them into their problem categories, the rest of this subsection will discuss how the models that are inferred by competitors are assessed, and how these are scored. The accuracy is assessed by comparing bit-strings produced by test cases on the inferred model against the bit-strings produced with respect to our reference models. The difficulty of the problem categories in Table 1 (important for determining the winning entry) is calibrated with respect to the performance of the baseline Blue-Fringe algorithm.

3.3.1 Measuring the Accuracy of an Hypothesis Model

Once a competitor has inferred a model, each test sequence in the test set is supplied to the model, to determine whether it is accepted by the model or not, producing a binary string to represent the classifications for the whole test set. To establish the accuracy of the model, this binary string is compared against the correct binary string produced by the hidden target model. The overlap between the two strings is measured with a measure called the *harmonic Balanced Classification Rate (BCR)*.

BCR is chosen for this competition because it has two important advantages over the standard error-rate measure. The standard approach of counting the proportion of correctly classified test sequences can be highly misleading if (a) the distribution of test sequences is imbalanced and (b) the target model happens to be imbalanced with respect to the proportion of sequences that it accepts or rejects (often the case with software models (Walkinshaw et al. 2008)). The BCR measure accounts for this imbalance by taking an average of two measures (elaborated below), which account for the abilities of a model to both accept the sequences it should, whilst correctly rejecting those that it shouldn't (sensitivity and specificity). Not only does this produce results that are more reliable, but it also enables us to scrutinise the results in more detail, with respect to its two base-measures (c.f. the bag plots in Figs. 4, 5, 6 and 7).

The standard BCR combines two factors. The sensitivity SE is the proportion of positive matches that are predicted to be positive. So in terms of the sets true positives (TP) and false negatives (FN), $SE = \frac{|TP|}{|TP \cup FN|}$. The specificity SP is the proportion of true negatives that are predicted to be negative, so $SP = \frac{|TN|}{|TN \cup FP|}$. The standard BCR is simply computed as the arithmetic mean of SE and SP.

To prevent a massive disparity between sensitivity and specificity from skewing the score, we choose to compute the BCR as the harmonic mean. This is computed as:

$$BCR = \frac{2 * SE * SP}{SE + SP}$$

A model is considered to be accurately inferred from a sample (or solved) if the harmonic BCR computed from the hypothesis model on the test strings is at least 99%. For each attempt by a competitor, the competition website returns a yes/no answer stating whether their model reaches this threshold. No further details on the actual performance are given to the competitor, to prevent hill-climbing on the test samples.

Table 2 Calibrating the mean BCR scores for a given cell, based on the scores of the benchmark Blue-Fringe algorithm shown in Table 1

Difficulty level	Score
1	$0.9 \leq score \leq 1$
2	$0.7 \leq score < 0.9$
3	$0.6 \leq score < 0.7$
4	$0 \leq score < 0.6$

3.3.2 Calibration

Table 1 gives rise to two important questions: (1) How many strings are required for a 100% sample, and (2) how do we find out how difficult each cell actually is? Both of these questions are answered by referring to what was considered to be the best inference technique before this competition—the Blue-Fringe algorithm (Lang et al. 1998) (see Appendix A). In doing so, the competition is being calibrated against the state-of-the-art, and any technique that succeeds to solve a cell can only do this by achieving some form of significant improvement. For the sake of calibration, and to provide potential competitors with a possible basis for developing their own algorithms, an implementation of the Blue-Fringe algorithm was produced in Ruby, and is available.[3]

The size of a 100% sample was calibrated by finding the training set for which the Blue-Fringe algorithm produces an accurate result for the simplest problems (i.e. the five models in top left cell of Table 1 where $|\Sigma| = 2$) without fully solving the cell.[4] Thus, any solution that manages to solve the cell will have to out-perform the baseline technique.

The difficulty of the cells was calibrated by establishing the average Blue-Fringe BCR score for each of the cells (shown in Table 1). Based on these averages, the cells are categorised into four levels from easy (1) to hard (4) according to the scheme in Table 2. These categories represent the number of points that are awarded to a technique for solving a given cell. The first algorithm to solve a cell was awarded the points, and the algorithm with the most points at the end of the competition was the winner. In the event that multiple algorithms achieved equal numbers of points (which turned out not to be the case) the first algorithm to have solved the harder cells would have been selected as the winner.

3.4 Running the Competition

The competition was run from March 2010, with the deadline for the final submission announced as the 31st of December 2010. It was advertised on software engineering mailing lists such as SEWORLD, and machine learning lists such as Google ml-news, and to authors from both the Software Engineering and Machine Learning communities who had previously developed DFA inference techniques were encouraged to participate. To incentivise the broader participation of challengers a prize of £700 was offered for the winning entry.[5]

[3] http://stamina.chefbe.net/baseline

[4] Although the average score for cell $|\Sigma| = 2$, 100% is 0.99, the Blue-Fringe algorithm did not achieve this performance for each of the five target models, and so did not successfully solve the cell.

[5] This was based on the value of the prize for the Abbadingo competition (Lang et al. 1998) which has been $1,024.

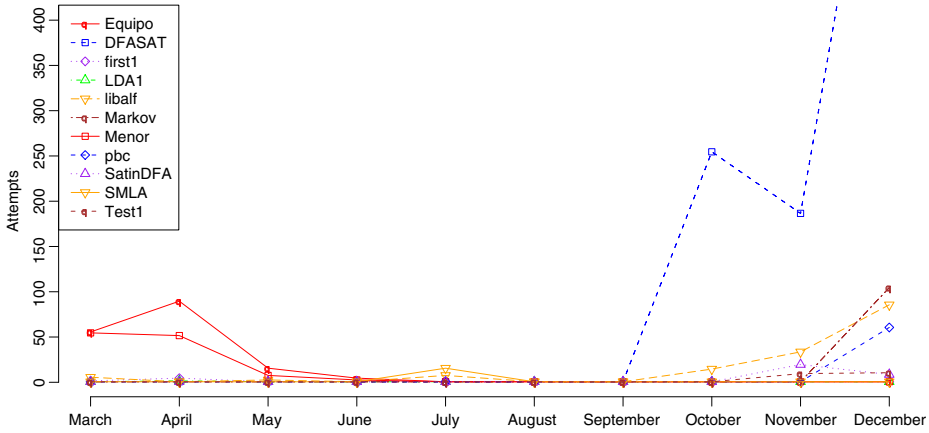

Fig. 1 Participation timeline—scale is capped at 400 submissions per challenger so as not to obscure lower participation levels on less competitive months. The DFASAT technique ultimately made 745 attempts in December

A Ruby implementation of the baseline algorithm was made available on the competition website (http://stamina.chefbe.net). The implementation is written in such a way that it is relatively straightforward to alter and extend. This was to encourage the involvement of people who did not have an existing framework of their own, and to provide them with a basis upon which to develop their own improved version.

The Google Analytics statistics (which should be interpreted with the usual caveats that accompany website statistics) suggest that the competition attracted a significant amount of interest. The StaMInA website was visited 3,885 times by 1,118 'unique' visitors. There were visits to the site from 69 countries. Of the top ten countries in terms of visits, seven were based in Europe, with significant visitor numbers from the US (ranked third), Brazil and Japan (ranked 9th and 10th).

Figure 1 shows the participation levels throughout the competition. It shows how, after an initial flurry of activity, there were four months of a small amount of sporadic activity, followed by a significant surge in activity before the deadline of the competition. By the end of the competition, 1,856 attempts had been made by eleven competitors. The names of the challenger techniques are given in the legend of the figure; the next section will discuss some of their individual performances in more detail.

4 Competition Results

Of the eleven competitors, five managed to solve problems that had not been solved by the baseline Blue-Fringe algorithm. Two of them (DFASAT and Equipo) managed to solve entire cells. In total, there were 61 successful attempts to solve problems, and competitors managed to solve a total of 43 distinct problems in 10 different cells.

The StaMInA hall of fame, of which a live version was kept on the website, is shown in Table 3 as it stood at the end of the competition. It shows that the DFASAT algorithm won by a significant margin.

From an empirical perspective, the competition is interesting because it provides an empirically sound basis for comparing different techniques against each other. There is no opportunity for competitors to bias the outcome of their techniques, and all are compared in a uniform way against the same success criteria. Furthermore, the competition provides a useful basis for collating data about the different techniques. Competitors submit their bit-strings for the respective problems, and although they only get to see the table shown in Table 3, the bit-strings provide a useful basis for gaining more detailed insights into the respective strengths and weaknesses of their techniques.

The box plots in Fig. 2 show the relative distributions of BCR scores for the various participating techniques, with respect to the four difficulty levels. Although the difficulty of the problems is in the same broad category, there is a degree of internal variance (inferring a hidden model with an alphabet of 25 might be harder than inferring a model with an alphabet of two). Furthermore, there is a major variance in the extent to which different techniques have been used to solve different cells in the category (the number of attempts is also reported in Fig. 2). As a result, these plots can only be treated as rough indicators of the comparative performances of the techniques. Level 4 needs to be treated particularly carefully; there were very few submissions, and several techniques (e.g. pbc) only attempted to solve cells that remained virtually untouched by the other competitors. Despite this caveat, the relative performance of those techniques that attempted a large number of cells adheres to a pattern that is broadly consistent across the different difficulty levels.

The remainder of this section will discuss some of the key techniques, and provide an overview of the main results. To investigate potential relationships between competing techniques, it is possible to resort to the data collected during the competition (in the form of bit-strings submitted by the competitors for the different techniques). This illustrates how the general mechanism of a competition is not only valuable from a perspective of prompting the development of new approaches (as will be seen with DFASAT), but also shows how it forms a valuable basis for gathering empirical data to gain insights into their performance.

In certain cases it is useful to compare the performance of two techniques in more detail than simply in terms of their BCR scores. In fact, a distribution of BCR scores can be represented in two dimensions in terms of sensitivity against specificity (of which the BCR score is the harmonic mean). When comparing the BCR distributions

Table 3 StaMInA hall of fame. Solved cells are annotated with the winning technique

$\|\Sigma\|$	Sparsity			
	100%	50%	25%	12.5%
2	Equipo (1)	*4 solved* (1)	– (3)	– (3)
5	DFASAT (1)	*1 solved* (2)	– (4)	– (4)
10	DFASAT (1)	*3 solved* (3)	– (4)	– (4)
20	DFASAT (1)	*4 solved*(3)	– (4)	– (4)
50	DFASAT (2)	DFASAT (3)	– (4)	– (4)

In any unsolved cells where individual problems had been broken, the number of solved problems is listed. The difficulty of each cell is included in parentheses, as based on Table 1

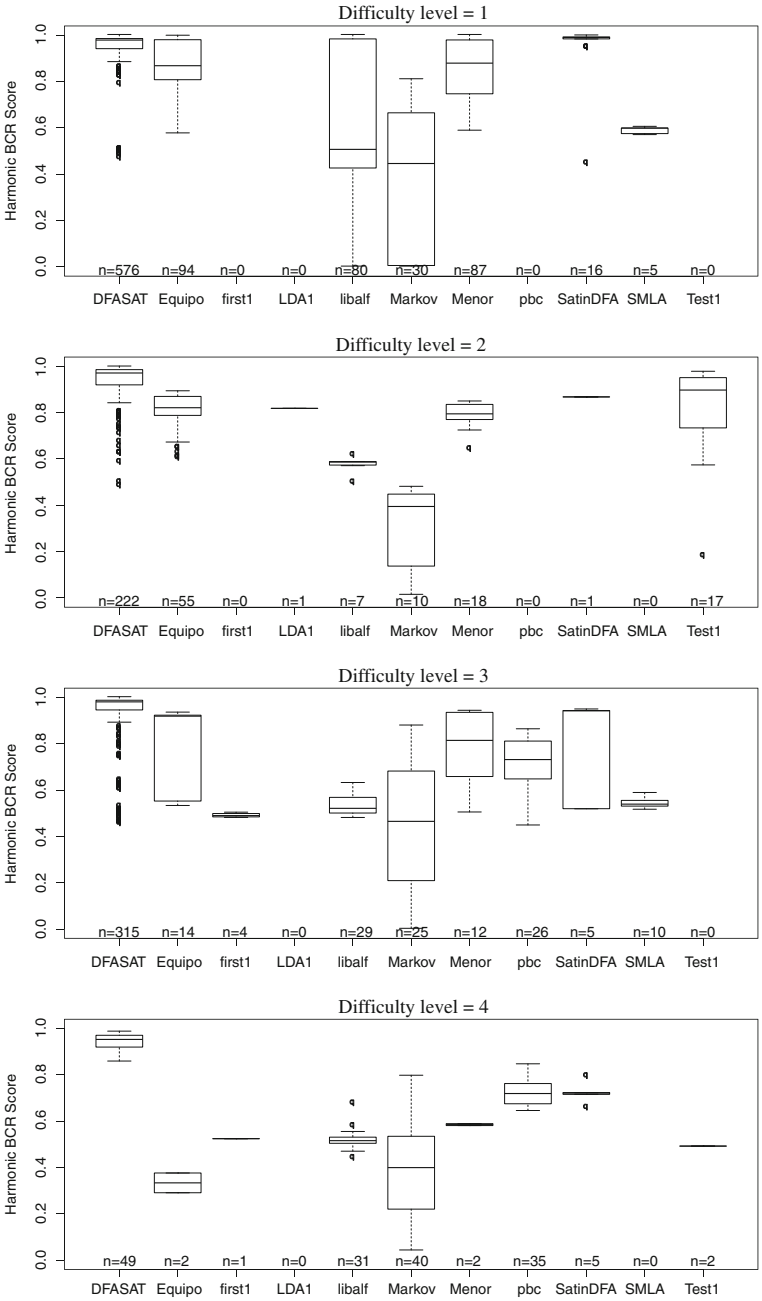

Fig. 2 *Bar plots* comparing performance of all algorithms according to difficulty level (see Tables 1 and 3). The number of attempts is noted under each box

for two different techniques, it is helpful to make the main concentrations of points explicit (as a box-plot does in single dimensions). To do this, we use bag plots (Rousseeuw et al. 1999), which are in effect two-dimensional analogs to box plots.

The bags centre upon the "median" point of the data, which is calculated as the point with the highest "halfspace depth" (Tukey 1975). The halfspace depth of a point p relative to a dataset P within a two-dimensional scatter plot is the smallest number of data points in P lying in any closed half-plane determined by a line drawn through p. In a bagplot, the inner bag contains the 50% of points with the highest depth. The outer fence is constructed by inflating the bag and taking the convex hull of the points inside the inflated bag, in analogy with the method for obtaining the whiskers in the box plot. Within the inner bag there is a further zone to indicate the 95% confidence region for the depth median of the set of points. The plots were produced with the bagplot function in the aplpack library[6] for the R statistical package.

4.1 Successful Techniques

4.1.1 DFASAT

The competition was ultimately won by the DFASAT algorithm, developed by Sicco Verwer and Marijn Heule. It solved all of the cells for 100% sparsity, and solved one of the cells for 50%. The algorithm is consistently the strongest performer, with both quartiles above 90% accuracy for all difficulty levels. This is even the case for difficulty level four where, with accuracy results that were mostly above 90%, it came very close to solving those cells that it attempted.

The DFASAT technique is based on previous work by the competitors (Heule and Verwer 2010). A detailed description of the algorithm and its adaptation for StaMInA is available in an accompanying paper (Heule and Verwer 2012), so it will be only summarised in abstract terms here. The essential idea that underpins the technique is to recode the DFA inference problem as a constraint satisfaction problem (and so to take advantage of many well-developed constraint satisfaction mechanisms). They achieve this by adopting a technique first proposed by Coste and Nicolas (1997), namely by encoding the inference challenge as a graph colouring problem, which can be solved reasonably efficiently with SAT solvers. This is however subject to scalability constraints, so the authors address this by first using a slightly modified version of the standard baseline technique (see Appendix A) to generate a partial solution, and then use the SAT technique to home-in on the exact DFA.

For the StaMInA competition, the original published DFASAT technique had to be altered in two significant ways (the competitors have noted that, in its orignal form, it was incapable of solving cells beyond the lowest difficulty level). The first alteration was to change the algorithm to take advantage of the characteristics of the StaMInA models. The way state pairs are chosen was changed from the traditional approach used in the baseline approach, taking advantage of the fact that large alphabets mean that any pair of states that have any outgoing transitions with identical symbols are much more likely to be equivalent than would be the case with smaller alphabets. Furthermore, the decision procedure used to select pairs of states to merge was perturbed stochastically. As a consequence, several candidate automata could be inferred for a single problem, and a heuristic was used to select the best candidate. This took into account knowledge about the likely size of the models

[6] http://cran.r-project.org/web/packages/aplpack/index.html

(about 50 states). Further details are available in the accompanying paper by Heule and Verwer (2012).

One outcome that might initially appear curious is the fact that, as shown in Table 3, DFASAT managed to solve the cell for an alphabet of 50, with a sample of 50%, but did not succeed in solving the seemingly easier cells for smaller alphabets in the same column. It is however important to bear two facts in mind. First, DFASAT still performed very strongly in the other cells; it was the only challenger to solve any problems in the column, and ultimately managed to solve 17 out of the 25 possible problems. Second, effort was not invested uniformly into solving each cell. The DFASAT algorithm employs heuristics to take advantage of large alphabets (this is discussed in the next subsection), and those cells with large alphabets are also associated with higher difficulty ratings, so their solution would lead to a greater reward for the competitors.

Having concluded the competition, the DFASAT authors ran their algorithm on the complete set of problems and cells. This performance is provided in Fig. 3. It shows clearly how, at every level of sparsity, the DFASAT comprehensively outperforms the Blue Fringe algorithm. One point of interest is that the performance of DFASAT does not deteriorate with higher alphabet sizes (as is the case with Blue Fringe). On the contrary, the charts clearly indicate that DFASAT excels at inferring models with a higher alphabet.

4.1.2 Equipo and Menor

The Equipo and Menor techniques are both based on the notion of *automata teams* (García et al. 2010). This approach is more closely aligned to the baseline Blue-Fringe technique than DFASAT. It uses the same Blue-Fringe approach (see Appendix A) to identify possible pairs of states to be merged. However, at this point it differs in a significant way. Instead of always choosing a specific pair based on a specific heuristic, pairs are chosen randomly. In this way, the algorithm can be executed several times on the same sample, and produce several different candidate DFAs. In this way, 'teams' of automata can be produced. Given a test set, each test sequence

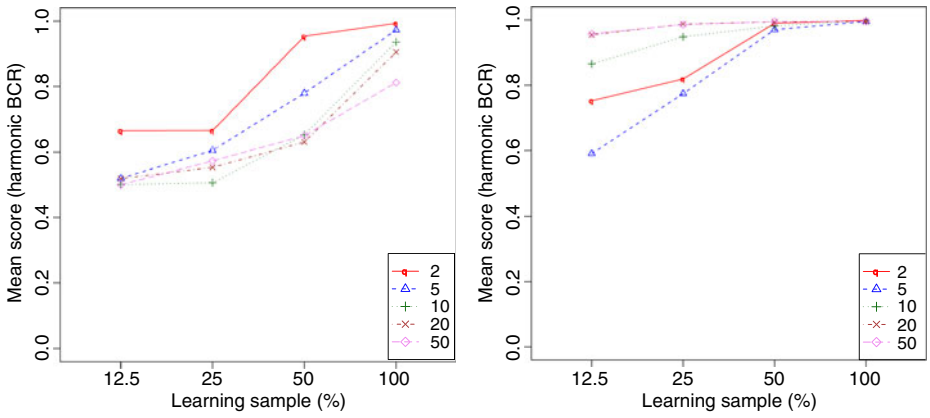

Fig. 3 Post-competition average benchmark performance of DFASAT (*left*) on all problems by sample sparsity, compared to equivalent performance by the baseline Blue-Fringe algorithm (*right*)

can be classified by arranging a vote within the team. One way to vary these algorithms (and perhaps the way in which Equipo and Menor differ) is to specify different voting strategies.

Judging from the box plots in Fig. 2, Equipo and Menor perform similarly to each other, achieving a reasonable accuracy for the first three difficulty levels. In such situations where the performance of two techniques is difficult to distinguish, the wealth of data collected during the competition becomes a valuable basis for carrying out a more detailed analysis. This is shown in the bag plot in Fig. 4. In the case of Equipo and Menor, the bag and fence corresponding to Menor are almost entirely subsumed by their Equipo equivalents. This indicates that, although their overall performance is effectively the same, the accuracy of Menor has a lower variance.

4.1.3 pbc

The pbc submission (which stands for Pattern Based Classification) by David Lo and Leonardo Mariani was one of the few competitors to have been based on a technique that had been specifically developed with a software-engineering application in mind. The underlying technique was developed by Lo et al. (2009), to infer software models from program traces. The approach also stands out from other techniques because it is the only approach to have been reasonably successful (see Fig. 2), yet is not a variant of the state merging algorithm that underpins the other approaches and the baseline approach. It is based on a kernel method, which applies statistical analysis to the given set of sequences to derive a model that can discriminate between valid and invalid sequences.

In terms of performance, pbc cannot be compared extensively to other competitors, because it was the only significant challenger for the cells it attempted (this highlights a danger in interpreting the broader boxplots in Fig. 2). The pbc challengers especially focussed on the six hardest cells ($|\Sigma| \geq 10$, sparsities of 25% and 12.5%). Given their relatively late entry into the competition (see Fig. 1), it is possible that this was due to the strategic aim of leapfrogging the other challengers instead of concentrating on the easier cells. Although the technique did not solve any of the problems in these cells, its accuracy was reasonably strong. This is shown in Fig. 5. The mean sensitivity is 0.71, and the mean specificity is 0.72, which is a

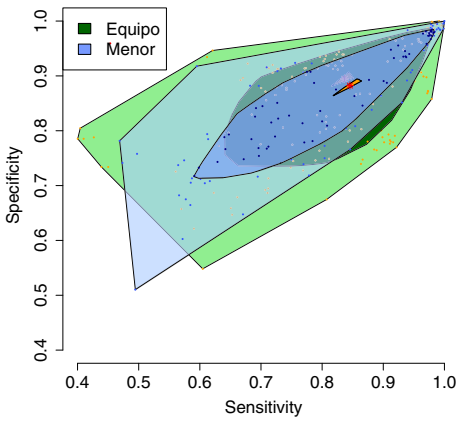

Fig. 4 Plot of Sensitivity versus Specificity for the Equipo and Menor techniques. Menor (*lighter bag*) is overlayed onto Equipo (*darker bag*). Note that the plot is rescaled

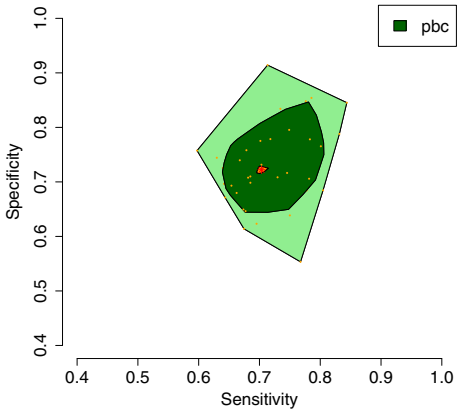

Fig. 5 Plot of Sensitivity versus Specificity for the pbc technique. Note that the plot is rescaled

major improvement on the baseline technique for those cells. In the light of the post-competition data from the DFASAT technique, the BCR performance of pbc is however still an average of 0.21 lower than the DFASAT technique.

4.1.4 SatinDFA

The SatinDFA technique only made a small number of submissions, but performed consistently well. Its performance across problems of difficulty level one is shown in Fig. 6. Of course it is important to bear in mind that there are only 16 SatinDFA submissions, against 576 submissions by DFASAT. Nonetheless, it is striking that the range of accuracy for SatinDFA is tightly focussed around the 0.98 in both dimensions.

SatinDFA was developed by Adriaans and Jacobs (2006) and Mulder et al. (2009). As with most of the other competitors, it is roughly based on the state merging process used by the Blue-Fringe technique (Appendix A). However, it differs from the baseline process in the strategy it uses to decide whether or not to merge a pair of states, which is based on the notion of a Minimum Description Length (MDL) Rissanen (1983). Since the computation of solutions using this heuristic is heavily

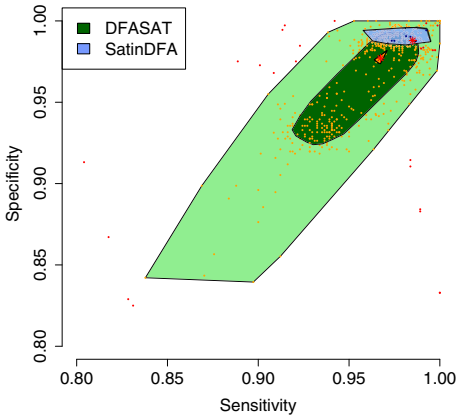

Fig. 6 Bag plots of Sensitivity versus Specificity for the SatinDFA technique (*light bag*) superimposed on the DFASAT technique results (*dark bag*) for difficulty level one. Note that the plot is rescaled

resource consuming, the SatinDFA technique is implemented with the Satin grid computation infrastructure (van Nieuwpoort et al. 2010), a Java framework that enables resource-intensive computations to be distributed accross a grid.

4.1.5 Test1

Judging by the boxplots in Fig. 2, the Test1 algorithm performed well in the cells it competed in (which were mostly at a difficulty level two). Looking at the bagplot in Fig. 7, this shows that the algorithm is distinctive in terms of its performance. There is a significant variance in terms of its sensitivity, but almost no variance in terms of its specificity. The standard deviation for sensitivity is 0.258, and is only 0.09 for specificity. This indicates that the inferred models can be susceptible to the tendency to reject too many sequences that ought to be accepted.

The algorithm was developed by James Scicluna and, appropriately for this competition, is specifically developed for DFAs with large alphabets. It differs from the baseline algorithm in the way that it compares states. The underlying idea is that a relatively long sequence of symbols is unlikely to be the suffix for more than one state. In other words, if two states in the prefix tree acceptor (see Section 2.3) share a relatively long suffix, they are likely to be equivalent and can be merged. The inference technique simply merges those states with matching suffixes that contain matching n-grams (according to the author, $n = 3$ worked best in this case). The algorithm is a version of the k-tails approach proposed by Biermann and Feldman (1972). This explains the relatively poor sensitivity results discussed above—one of the widely acknowledged problems with the k-tails is that it will fail to merge states that are equivalent but happen to not have matching k-tails (Reiss and Renieris 2001), producing models that can falsely reject a large proportion of sequences. Nevertheless, the basic criterion for selecting states was still relatively successful, producing a reasonably high BCR score.

4.2 Attributes of Successful Inference Techniques

The competition has a much broader aim than simply to select a specific winning technique. Several of the participating techniques excelled in particular areas of the competition. To supplement what has so far been a purely quantitative perspective

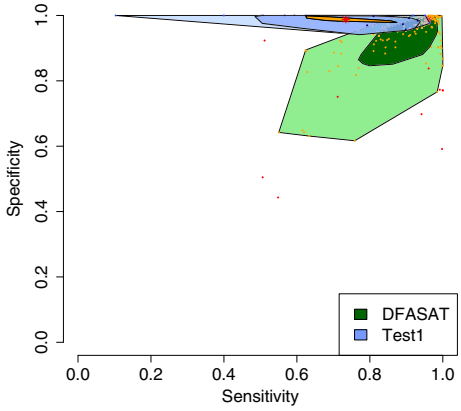

Fig. 7 Bag plots of Sensitivity versus Specificity for the Test1 technique (*light bag*) superimposed on the results for the DFASAT technique (*dark bag*) for difficulty level two

on the accuracy of techniques, this subsection provides an overview of a selection of factors that potentially played a role in the success of the algorithms.

Stochasticity and Hypothesis Diversity Several contributors to the competition (DFASAT, Equipo and Menor), all involve a degree of randomness, to generate populations of potential solutions to the same problem. This mitigates the risks that are involved in conventional algorithms, such as the baseline Blue-Fringe algorithm, where any inadequate state merging early in the inference process will lead to a solution that is potentially highly inaccurate.

Encoding Model Inference in Formats that are Efficiently Computable Both DFASAT and SatinDFA are able to adopt inference procedures that would usually be prohibitively expensive from a computational viewpoint. This is achieved by encoding aspects of the grammatical inference problem in a format that can be solved more efficiently. DFASAT maps the original problem to a boolean satisfiability problem that can be solved efficiently by SAT solvers. SatinDFA involves what would conventionally be a prohibitively expensive merging heuristic, but is able to use this by using the Satin platform to evaluate several merges in parallel.

Alternatives to State Merging The success of the pbc algorithm in the harder cells of the competition demonstrates that there are powerful alternatives to state merging. Alternative techniques (such as the kernel-based approach used by pbc) can potentially complement the standard state-merging approaches used by the other techniques.

Exploiting Characteristics of the Target Models DFASAT, Test1, and pbc were particularly successful because of the large alphabets in the StaMInA models, a target characteristic that would be seen as detrimental to the performance of other traditional techniques. DFASAT exploited knowledge about the rough size[7] of the target models to reduce the search space. Test1 and pbc both took advantage of the frequency of particular sequences, which becomes a more reliable indicator of equivalent states as the alphabet increases in size.

5 Discussion

This section will look at the threats to validity that arise when interpreting the StaMInA results. This will be followed by some suggestions for future competitions.

5.1 Threats to Validity

The competition has successfully encouraged the development and adaptation of techniques to infer models of software systems. However, on top of this, the argument is made that the competition can be seen as a form of experiment, to make

[7]The number of states of the target models was undisclosed but known to be approximately equal to 50, with actual values between 41 and 59 states.

empirically sound decisions about the relative accuracy of the various techniques. This has to however be qualified; any conclusions that are drawn from this data are subject to threats to validity.

5.1.1 Representativeness of the Models

The target models used in the StaMInA competition are meant to be broadly representative of typical software models. Given the diverse nature of software systems, and the different levels of abstraction at which they might be modelled, the notion of what constitutes a "typical software model" is very difficult to pin down. Any algorithm that claims to produce realistic software models is going to be very difficult to validate for the same reason.

Even though the realism of the models is difficult to validate in a formal sense, their key characteristics (large alphabets and non-uniform transition structures) are typical of software models and are widely acknowledged. It is therefore reasonable to conclude that an inference technique that is capable of inferring these models is likely to be capable of inferring other non-synthetic models of software systems.

5.1.2 Representativeness of the Traces

The traces from which models are inferred are generated according to a process that enables one to carefully control the amount of information about the target model that is contained within a given set of traces. Although this ensures internal validity from an experimental point of view, it leads to a possible question mark over the external validity. There is a danger that a technique might excel at inferring accurate models from StaMIna traces, but fail to perform as well (relatively to competing techniques) on software traces in a specific application domain.

There is a trade-off between the control exercised over the information content of the traces, and their representativeness with respect to software engineering applications. Ensuring that there are even proportions of positive and negative sequences, and generating negative sequences as mutations of positive ones will invariably lead to distributions of traces that could be construed to be artificial and not representative of "typical" traces that arise in software engineering application domains. However, this provides the benefit of being able to carefully monitor the performance of a technique with respect to varying degrees of information contained in the traces.

It is unclear how the trace-generation approach might be changed to become more reflective of general software engineering tasks, due to the inherent diversity of the types of traces they use. For example, biasing the approach to produce a greater proportion of positive trace would be more representative of certain areas (e.g. passive testing and runtime verification), but less representative of others (e.g. debugging from failure traces). The current selection is deliberately application-neutral. Accordingly, the implications for the conclusions about algorithm performance are best interpreted on the understanding that certain algorithms might excel for different selections of traces.

5.1.3 Intervention by Competitors

One factor that cannot be controlled in a competition setting is the behaviour of the competitors. Invariably, if a technique fails to work in the first instance, there will be

a tendency to alter its behaviour tweaking its parameters to better suit the training samples or target model characteristics. This means that, when interpreting the data collected throughout the competition, it will include the results of techniques that were produced throughout. For example, the winning DFASAT entry required a reasonably substantial amount of alteration to successfully solve the harder cells in the competition. However there is no way of distinguishing between different 'phases' of the technique in the raw results data. The only way for a participant to enable this distinction is to submit different configurations of their technique as separate entries into the competition, as happened with the Equipo and Menor entries.

5.1.4 Strategic/Uneven Participation

One factor that hampers the comprehensive comparison of different techniques is the fact that competitors concentrated their efforts on different parts of the grid. For example, from an empirical perspective it would have been nice to directly compare the performances of DFASAT and pbc, however neither technique attempted any of the same cells during the competition, although they performed well independently. In some cases the choice to focus on particular cells would have been a strategic choice by the competitors. In other cases, participants seemed to give up, under the (often false) perception that their technique was not performing very well because it failed to hit the 99% accuracy mark for any.

This means that broad comparisons of the competition data alone, such as the box-plot view shown in Fig. 2, have to be interpreted with some caution. When carrying out more detailed comparisons, as with the bag plots, care has been taken to ensure that the techniques at least competed for the same cells. To facilitate a more in-depth analysis of the performances, all competitors were invited to rerun their techniques on all of the problems after the competition had completed. One competitor that did volunteer to do so was the winning DFASAT team. Their performance (as charted in Fig. 3) can thus be confirmed to have outperformed the other competitors on all of the cells.

5.2 Current Use

Although the formal competition has been completed, the StaMInA website is being maintained, to act as a benchmark platform for the evaluation of future techniques. This will also enable those competitors who had given up under the false impression that their techniques performed poorly to attempt the rest of the cells.

One lesson that was drawn from the competition is that the lack of feedback to the competitor, designed to prevent hill-climbing, was ultimately perhaps too opaque. If competitors failed to obtain a score of 99%, the only feedback they obtained was that their technique had failed. This will lead to despondency for competitors whose techniques fail to reach this high margin, and seemed to prevent several StaMInA competitors from seriously attempting cells that they might have eventually succeeded in solving.

For a conventional benchmarking system, the absence of a prize means that there is less of an incentive for participant to cheat. As a consequence, the feedback system has been opened up, and now provides the conventional BCR score on a per-problem basis. This also addresses one of the threats to validity listed above, by encouraging competitors to attempt all of the problems in a given cell/difficulty level.

6 Related Work

To the best of the author's knowledge, there have been no competitions in the past to drive software engineering research. Competitions have however been used to spur the development of novel techniques in numerous other research areas. One popular example from the field of bioinformatics is the Critical Assessment of Techniques for Protein Structure Prediction (CASP) competition series,[8] a biennial competition series that has become one of the main drivers for the development of protein structure prediction techniques. In the field of automated reasoning, the CADE ATP System Competition (CASC)[9] has been a successful series of competitions to compare the performances of automated theorem provers.

6.1 State Machine Inference Competitions

At its heart, the StaMInA competition is a state machine inference competition. There is an intrinsic relationship between state machine inference and grammatical inference. The StaMInA fits the mould of numerous grammatical inference competitions that have been arranged over the past decade. However, this competition is specifically intended for learning models typically found in the Software-Engineering domain.

The Abbadingo-One competition (Lang et al. 1998) took place in 1997, and gave rise to the Blue-Fringe algorithm, which is used as the baseline technique in this competition. This was followed by the Gowachin noisy DFA inference competition in 2004, which was followed up by the similar GECCO 2004 noisy DFA inference competition.[10] The Gowachin and the GECCO competition however involve a step that adds noise to the training data. As mentioned in the background, StaMInA differs from these competitions in three ways: (1) the target models have larger alphabets, (2) the testing and training samples are derived directly from the target model, and (3) results are evaluated with the BCR metric as opposed to simply counting the proportion of correct classifications.

The ZULU competition[11] (Combe et al. 2010) is a recent competition for the inference of DFAs, however this is developed for the active learning setting—i.e., where there is an oracle present that is capable of answering questions from the inference technique. Unlike StaMIna, ZULU also reuses the usual target model generation algorithm from Abbadingo-One or Gowachin.

6.2 Benchmarking in Software Engineering Research

One area that has become increasingly prevalent in empirical software engineering research is the use of benchmarks. In essence, a benchmark consists of a collection of subject data that can be consumed by different techniques, and can be used to draw coherent and valid conclusions about the respective performance of these techniques. The case for benchmarking is discussed comprehensively by Sim et al. (2003).

[8] http://predictioncenter.org/

[9] http://www.cs.miami.edu/~tptp/CASC/

[10] http://cswww.essex.ac.uk/staff/sml/gecco/NoisyDFA.html

[11] http://labh-curien.univ-st-etienne.fr/zulu/

The StaMInA competition can be seen as a form of benchmark. According to Sim et al.'s definition of a benchmark, it should compare relevant aspects of techniques, the data used to compare techniques should be representative of realistic tasks, and it should use well-defined measurements of performance. These three aspects all apply to the StaMInA competition.

Perhaps one aspect that stands out with respect to competitions is the fact that, by the nature of a competition, it encourages wide-spread participation in the way that a conventional benchmark might not. Competitions offer an incentive to participate (the offer of a prize, and the potential to top the hall of fame). This not only broadens the participation, but also spurs the development of new techniques to tackle the problem at hand. This has been evident in the StaMinA competition, where competitors actively attempted to tune and improve their techniques to outperform others.

7 Conclusions and Future Work

It is widely acknowledged that the widespread empirical comparison of software engineering techniques can be exceptionally difficult. Tool-support for techniques is often poor, tools tend to be poorly maintained, are difficult to obtain, or configure, and can often be too expensive to re-implement. This is further exacerbated by the challenge of encoding the subject data into a suitable format, and to obtain coherent results. Competitions such as StaMInA circumvent this problem, by providing incentives for the participants to run their own tools, and so removing these major practical issues.

The competition was successful at motivating the participation of challengers, who contributed several techniques that mark a significant improvement on the state of the art. The DFASAT technique, which was altered to suit the competition and ultimately won, represents a major step forward in the use of software model inference techniques. Were a new competition to be held in the area, it would represent the new base-line algorithm for other challengers to improve on.

From an empirical software engineering viewpoint, the StaMInA competition is interesting. Software techniques are difficult to compare in a systematic way, because their tools (often prototypes) are often difficult to obtain or execute. The competition removes these restrictions by placing the onus on the developer. It also provides an incentive for widespread participation by offering participants the opportunity to directly compare the performance of their techniques against the state of the art, and against other techniques.

As a legacy, the StaMInA website has been converted into a standard benchmarking site. The actual models and test labels remain undisclosed, but each participant can obtain the actual BCR scores for their technique (as opposed to the Boolean correct/incorrect feedback that was used during the competition). This is in the process of being expanded, to provide a more detailed hall of fame, along with more detailed performance indicators for the top three techniques.

As far as future work is concerned, there are several ways in which the StaMInA framework could be extended or improved for future competitions. One point of interest would be to consider more advanced types of target models. For example, probabilistic state machines are of particular interest, not least because they can be inferred from samples that contain only positive examples, which are much easier to

obtain with respect to software systems. Models with a data state such as Extended Finite State Machines (EFSMs) are another interesting, albeit more challenging prospect for use as targets. Besides extending the type of model under consideration, there is also the potential to consider different domains. As an example, there are several biological datasets that would fit this competition framework.

Acknowledgements The authors sincerely thank all of the competitors who participated in the StaMinA competition, as well as the members of the competition steering committee. Several contributors subsequently provided useful insights about their techniques/algorithms. These include Sicco Verwer at KU Leuven and Marijn Heule at T/U Eindhoven (DFASAT), Damian Rodriguez, Pedro Garcia and Manuel Vasquez at UP Valencia (Equipo/Menor), Ceriel Jacobs at VU Amsterdam (SatinDFA), David Lo at SMU, Singapore and Leonardo Mariani at Milan Bicocca (pbc), and James Scicluna at the University of Malta (Test1). Finally, the authors thank the reviewers for their thoughtful and constructive feedback.

Appendix A: The Baseline Blue-fringe Algorithm

The ultimate aim of the competition is to find algorithms that improve on the state of the art. To ensure this, the competition is calibrated with respect to what was believed to be the best Algorithm before the competition—Price's Blue-Fringe algorithm , which was the winner of the Abbadingo One grammatical inference competition and is described by Lang et al. (1998). This subsection gives a brief overview of the algorithm, to provide a context to the challenge presented by the competition. The reader can refer to Lang's original paper for more details.

Algorithm 1: Basic State Merging Algorithm

Input: Two samples S^+ and S^- containing positive and negative examples respectively
Result: A DFA consistent with S^+ and S^-

```
1  Infer(S+, S−) begin
2      PTA ←initialize(S+, S−);
       // Let N denote the number of states in the PTA
3      π ← {{0}, {1}, ..., {N − 1}};
4      while (Bi, Bj) ← ChoosePair(π) do
5          πnew ← Merge(π, Bi, Bj);
6          if Compatible(PTA/πnew, S−) then
7              π ← πnew;
8      return PTA/π

   // Function to merge pair of states and ensure that the result is
      deterministic
9  Merge(π, Bi, Bj) begin
10     π ← π \ {Bi, Bj} ∪ {Bi ∪ Bj};
11     while (Bk, Bl) ←FindNonDeterminism(π, Bi, Bj) do
12         π ←Merge(π, Bk, Bl);
```

The basic state-merging algorithm is shown in Algorithm 1 (from Lambeau et al. 2008). The basic process of the algorithm can be explained as follows:

1. (line 2) Initialise the PTA from $S^+ \cup S^-$.
2. (line 3) Initialise the partition of the set of states in the PTA (denoted π), such that each state has its own block in the partition.
3. (lines 4–7) In the loop, pairs of blocks are selected iteratively, following the strategy specified by the `ChoosePair` function. The choice of state pairs is key to the accuracy of the final model, and it is the strategy employed in `Choosepair` that distinguishes state-merging algorithms from each other. The strategy employed by Blue-Fringe is discussed briefly below.
4. (line 5, 9–12) Every time a pair of states are selected, they are merged using the `Merge` function. This removes the two individual blocks from π and replaces them with a single super block. When applying this set union to the PTA, it entails the removal of the two states represented by the blocks, and replacing them with a single state with all of the incoming/outgoing transitions of the two individual states.
5. (line 11–12) If the two blocks have outgoing transitions that share the same labels, merging them will produce a non-deterministic state machine. This is addressed by recursively calling the `Merge` function to merge the blocks that are the targets of the respective non-deterministic transitions until the machine is deterministic.
6. (line 6) By construction, any DFA that is the product of a merge will still accept every example in S^+. However, it can occur that it also wrongly accepts examples in S^- (this is known as *overgeneralisation*). To ensure that such merges do not contribute to the final inferred model, the call to the `Compatible` function in line 6 checks that the DFA that is produced by merging the states in PTA according to π_{new} (denoted PTA/π_{new}) still rejects the negative examples in S^-.
7. (line 7) If the merged hypothesis is compatible, the current state partition π is updated to reflect the updated hypothesis π_{new}.
8. (lines 4, 8) Once `ChoosePair` cannot produce any more pairs of blocks, the final state machine can be returned by applying the partition π to the PTA (denoted PTA/π in line 8).

The Blue-Fringe algorithm is distinguished from other algorithms by its specific implementation of the `ChoosePair` function. It is practically impossible to assess every possible set of merges – the number of partitions of the set of states in the PTA is computed by the Bell number of the number of states in the PTA, which rises exponentially as the size of the PTA increases. The trick of the Blue-Fringe algorithm is to only consider a small fraction of the possible merges, but to select the candidate state pairs according to a strategy that is most likely to yield pairs that are in fact equivalent.

The Blue-Fringe version of `ChoosePairs` works from the root of the PTA outwards. At each iteration of `Infer`, it considers a pool of possible candidates pairs (Lang et al. 1998). This pool is determined by identifying the set of states that cannot be merged with each other (coloured red), and the set of states that are adjacent to these red states (coloured blue). This is illustrated in Fig. 8. The pool of candidate merges consists of every possible red-blue pair of states.

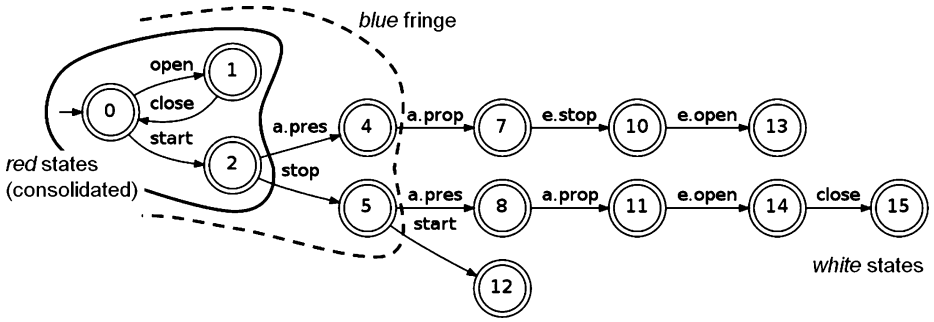

Fig. 8 Illustration of Blue-Fringe state selection

The best pair is chosen by scoring each pair (computed by measuring the extent to which the outgoing set of paths from each state overlap). The rationale for this heuristic is to reduce the likelihood of merging a pair of states that are actually not equivalent by concentrating on those pairs for which there is a greater amount of evidence that they are actually equivalent. Once a pair has been merged, the red-blue sets are recomputed and the process continues with the new pool of candidate pairs.

Appendix B: Random DFA Generation Algorithm

A random DFA must obey several constraints. Every state must be reachable from the initial state, it must be deterministic (no state can have two outgoing transitions with the same label) and minimal (no two states can be equivalent). On top of that, it must be possible to generate a population of those machines that obey the characteristics listed above.

The observation that software models tend to have an uneven distribution of in-/out-degrees and a small number of hubs and authorities is significant. This is a feature of a family of directed graphs that are generally referred to as "complex networks" (Kleinberg 1999; Leskovec et al. 2007), which are commonly used to model networks such as the world-wide web and social networks. The observation implies that algorithms for the generation of random complex networks (which constitute a significant part of complex network research) make a reasonable starting point for the generation of random state machines.

The Forest-Fire algorithm by Leskovec et al. (2007) has been shown to produce directed graphs that are especially suited for representing complex networks in a variety of domains. It is based on an iterative algorithm, where a new node is added at each iteration, and is connected to other nodes in the graph by selecting the closest connections of the first node it connected to. The rest of this section provides a brief overview of the forest fire algorithm, and shows how it has been adapted to produce state machines.

This description closely follows that of Leskovec et al. (2007), who can be referred to for further details. The algorithm has three parameters: a *forward-burning probability* f, a *backward-burning ratio* b, and the number of vertices n. Consider a node v

to be added to the graph at a time t where $0 < t \leq n$. Node v forms an edge to nodes in the graph at time t as follows:

1. Choose a random ambassador node $w \neq v$ and form an edge $v \rightarrow w$.
2. Generate two random numbers x and y that are geometrically distributed with means $f/(1-f)$ and $fb/(1-fb)$ respectively. Node v selects x out-edges and y in-edges of w to nodes that are not yet visited. If there are not enough nodes available, it selects as many as it can.
3. v forms out-edges to the end-points of the selected edges from and to w and applies step (2) recursively for each of those nodes. As the process continues, nodes cannot be revisited.

The above algorithm cannot be used as-is to synthesise state-machines. As new nodes are added, they are unreachable from any of the other nodes in the machine. By default, an edge cannot be added from one node to itself, ruling out self-looping states, which are common in software models. The original algorithm does not account for the notion of terminal states. Furthermore, a strategy is required to ensure that the state transitions in the final machine are suitably labelled (i.e., that the machine is deterministic and minimal). This needs to account for the fact that there could be multiple transitions between the same pair of states.

The following adaptations have been made to ensure that the final graph is state-machine like.

- **Alphabet** To add the transition labels, an additional parameter a is used, which is the upper limit on the size of a vector of numbers representing different elements of the alphabet. Every time an edge is added, it is labelled with a random element from that vector. The possible choices are curtailed to ensure that a selected element will not cause the machine to be non-deterministic. If there are no available alphabet elements left, the edge is not added.
- **State reachability** To ensure that each state is reachable, every time a state is added, instead of connecting an edge from the new state to an ambassador state, the reverse edge is added (from the ambassador to the new state).
- **Accepting states** Every time a state is added it is randomly labelled (with a probability of 0.5) as an accepting (final) state or not.
- **Self-loops** In step 2 of the conventional algorithm, it is impossible to add edges from a state to itself. We have added a parameter s to make it possible to specify the probability for this to occur.

To identify suitable parameter values for the algorithm, random DFAs were synthesised for a range of parameters f, b and the self-looping probability s. The alphabet-size parameter a only affects the structure of the machine for small alphabet sizes (i.e. $a = 2$), constraining states to an out-degree of 2. However, since this is a special case it is not factored in to the calibration of the main parameters. For every configuration, certain measurements were plotted, including the number of transitions, states, depths, and hub/authority distributions. These were compared to the equivalent plots from the real sample of machines. Several configurations produced reasonable looking DFAs, but the configuration $f = 0.31$, $b = 0.385$ and $s = 0.2$ resulted in measurements that were deemed to be most suitable in terms of the plots, as well as a manual inspection of the DFAs themselves.

References

Adriaans P, Jacobs C (2006) Using MDL for grammar induction. In: International colloquium on grammatical inference: algorithms and applications (ICGI). Lecture notes in artificial intelligence, vol 4201, pp 293–306

Ammons G, Bodík R, Larus J (2002) Mining Specifications. In: Principles of programming languages (POPL). Portland, Oregon, pp 4–16

Angluin D (1978) On the complexity of minimum inference of regular sets. Inform Control 39:337–350

Biermann A, Feldman J (1972) On the synthesis of finite-state machines from samples of their behavior. IEEE Trans Comput 21:592–597

Biermann A, Krishnaswamy R (1976) Constructing programs from example computations. IEEE Trans Softw Eng SE-2:141–153

Combe D, de la Higuera C, Janodet J (2010) Zulu: an interactive learning competition. In: Finite-state methods and natural language processing. Lecture notes in computer science, vol 6062. Springer, pp 139–146

Cook J, Wolf A (1998) Discovering models of software processes from event-based data. ACM Trans Softw Eng Methodol 7(3):215–249

Coste F, Nicolas J (1997) Regular inference as a graph coloring problem. In: Workshop on grammatical inference, automata induction, and language acquisition (ICML'97), pp 9–7

Damas C, Lambeau B, Dupont P, van Lamsweerde A (2005) Generating annotated behavior models from end-user scenarios. IEEE Trans Softw Eng 31(12):1056–1073

Dupont P, Lambeau B, Damas C, van Lamsweerde A (2008) The QSM algorithm and its application to software behavior model induction. Appl Artif Intell 22:77–115

Dupont P, Miclet L, Vidal E (1994) What is the search space of the regular inference? In: Proceedings of the international colloqium on grammatical inference and applications (ICGI'94). LNAI, vol 862. Springer Verlag, pp 25–37

García P, de Parga M, López D, Ruiz J (2010) Learning automata teams. In: International colloquium on grammatical inference: algorithms and applications (ICGI). Springer, pp 52–65

Gold M (1978) Complexity of automaton identification from given data. Inform Control 37:302–320

Heule M, Verwer S (2010) Exact DFA identification using SAT solvers. In: International colloquium on grammatical inference: algorithms and applications (ICGI), vol 6339. Springer, pp 66–79

Heule M, Verwer S (2012) Software model synthesis by indentifying DFAs using satisfiability solvers. Empir Software Eng (submitted)

Hopcroft J, Motwani R, Ullman J (2007) Introduction to automata theory, languages, and computation, 3rd edn. Addison-Wesley

Keller RM (1976) Formal verification of parallel programs. Commun ACM 19:371–384

Kleinberg J (1999) Authoritative sources in a hyperlinked environment. J ACM 46(5):604–632

Lambeau B, Damas C, Dupont P (2008) State-merging DFA induction algorithms with mandatory merge constraints. In: International colloquium on grammatical inference: algorithms and applications (ICGI). Springer, pp 139–153

van Lamsweerde A (2009) Requirements engineering: from system goals to UML models to software specifications. Wiley

Lang K (1992) Random DFA's can be approximately learned from sparse uniform examples. In: Proceedings of the international conference on learning theory (COLT'92), pp 45–52

Lang K, Pearlmutter B, Price R (1998) Results of the abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In: International colloquium on grammatical inference and applications(ICGI). LNAI, vol 1433, pp 1–12

Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines—a survey. In: Proceedings of the IEEE, vol 84, pp 1090–1126

Leskovec J, Kleinberg J, Faloutsos C (2007) Graph evolution: densification and shrinking diameters. ACM Trans Knowl Discov Data (or TKDD) 1(1)

Lo D, Cheng H, Han J, Khoo S, Sun C (2009) Classification of software behaviors for failure detection: a discriminative pattern mining approach. In: International conference on knowledge discovery and data mining (KDD2009). ACM, pp 557–566

Lo D, Khoo S (2006) SMArTIC: towards building an accurate, robust and scalable specification miner. In: Foundations of software engineering (FSE), pp 265–275

Lorenzoli D, Mariani L, Pezzè M (2008) Automatic generation of software behavioral models. In: ICSE '08: Proceedings of the 30th international conference on software engineering, pp 501–510

Magee J, Kramer J (1999) Concurrency: state models and java programs. Wiley

Mulder W, Jacobs C, van Someren M (2009) Collaborative DFA learning applied to grid administration. In: Benelearn, annual Belgian-Dutch conference on machine learning. Tilburg, the Netherlands, pp 38–46

van Nieuwpoort R, Wrzesińska G, Jacobs C, Bal H (2010) Satin: a high-level and efficient grid programming model. ACM Trans Progr Lang Syst 32(3):1–39

Oncina J, Garcia P (1992) Inferring regular languages in polynomial update time. In: Pattern Recognition and Image Analysis, vol 1. World Scientific, Singapore, pp 49–61

Raffelt H, Steffen B, Berg T, Margaria T (2009) Learnlib: a framework for extrapolating behavioral models. STTT 11(5):393–407

Reiss S, Renieris M (2001) Encoding program executions. In: International conference on software engineering(ICSE), pp 221–230

Rissanen J (1983) A universal prior for integers and estimation by minimum description length. Ann Stat 11(2):416–431

Rousseeuw P, Ruts I, Tukey J (1999) The bagplot: a bivariate boxplot. Am Stat 53(4)

Shahbaz M, Groz R (2009) Inferring mealy machines. In: FM 2009: formal methods, second world congress, Proceedings. Lecture notes in computer science, vol 5850. Springer, Eindhoven, The Netherlands, pp 207–222, 2–6 November 2009

Sim S, Easterbrook S, Holt R (2003) Using benchmarking to advance research: a challenge to software engineering. In: International conference on software engineering (ICSE), pp 74–83

Tukey J (1975) Mathematics and the picturing of data. In: Proceedings of the international congress of mathematicians, vol 2, pp 523–531

Walkinshaw N, Bogdanov K (2008) Inferring finite-state models with temporal constraints. In: International conference on automated software engineering (ASE)

Walkinshaw N, Bogdanov K, Holcombe M, Salahuddin S (2007) Reverse engineering state machines by interactive grammar inference. In: International working conference on reverse engineering (WCRE)

Walkinshaw N, Bogdanov K, Holcombe M, Salahuddin S (2008) Improving dynamic software analysis by applying grammar inference principles. J Softw Maint Evol: Res Pract 20(4)

Walkinshaw N, Bogdanov K, Johnson K (2008) Evaluation and comparison of inferred regular grammars. In: International colloquium on grammatical inference: algorithms and applications (ICGI). Lecture notes in computer science, vol 5278. Springer, pp 252–265

Neil Walkinshaw received a B.Sc. in Computer Science from the University of Sheffield in 2002, and a Ph.D. in Computer Science from the University of Strathclyde in 2006. He was subsequently a post-doctoral researcher with the Verification and Testing group at the University of Sheffield until 2010. Since 2010 he has been a lecturer in Computer Science at the University of Leicester. His research interests revolve around the activities of software verification and validation, and the use of inference and automated reasoning techniques to support these.

Bernard Lambeau received the MSc (2003) and PhD (2011) degrees in computer science from the Université catholique de Louvain. He is currently a postdoctoral researcher working with Axel van Lamsweerde. His research interests include grammar induction applied to model synthesis as well as requirements engineering, software design and development.

Christophe Damas received the MSc (2003) and PhD (2011) degrees in computer science from the Université catholique de Louvain. He is currently a postdoctoral researcher working with Axel van Lamsweerde. His research interests include requirements engineering, process modeling and analysis.

Kirill Bogdanov is currently a lecturer in the Department of Computer Science at the University of Sheffield. Bogdanov got his BSc from the Moscow Institute of Physics and Technology and a PhD from The University of Sheffield. Until his appointment as a lecturer in year 2000, he worked as a Research Associate on the "Method for Object Testing, Integration and Verification (MOTIVE)" project. His interests are in methods for rigorous state-based testing of software, aided by reverse-engineering in order to obtain state-based models. Testing is a part of exploration of a system being reverse-engineered. Recent work involves using state-based representation of domain constraints and type inference to improve the quality of reverse-engineered models.

Pierre Dupont received an M.S. in Electrical Engineering from the Université catholique de Louvain (Belgium) in 1988, and a Ph.D. in Computer Science from l'Ecole Nationale Supérieure des Télécommunications, Paris (France) in 1996. From 1988 to 1991, he was a research staff member of the Philips Research Laboratory Belgium. In 1992, he joined the France Telecom Research Center, Lannion (France). Its primary research was in automatic speech recognition with a special focus on search algorithms and language modeling. He has been visiting researcher at Carnegie Mellon University, Pittburgh (USA) in 1996–1997, and at Universidad Politécnica de Valencia (Spain) in 1994, 1995 and 2000. From 1997–2001, he was Associate Professor in the Computer Science Department of the University of Saint-Etienne, France. He is currently Full Professor at the Louvain School Engineering and co-founder of the Machine Learning Group of the Université catholique de Louvain, Belgium. His current research interests include machine learning with applications to computational biology and medicine, feature selection and biomarker discovery, statistical language modeling, grammar and automata induction, graph mining.