Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Software Model Synthesis using Satisfiability Solvers

Marijn J.H. Heule · Sicco Verwer

the date of receipt and acceptance should be inserted later

Abstract We introduce a novel approach for synthesis of software models based on identifying deterministic finite state automata. Our approach consists of three important contributions. First, we argue that in order to model software, one should focus mainly on observed executions (positive data), and use the randomly generated failures (negative data) only for testing consistency. We present a new greedy heuristic for this purpose, and show how to integrate it in the state-of-the-art evidence-driven state-merging (EDSM) algorithm. Second, we apply the enhanced EDSM algorithm to iteratively reduce the size of the problem. Yet during each iteration, the evidence is divided over states and hence the effectiveness of this algorithm is decreased. We propose – when EDSM becomes too weak – to tackle the reduced identification problem using satisfiability solvers. Third, in case the amount of positive data is small, we solve the identification problem several times by randomizing the greedy heuristic and combine the solutions using a voting scheme. The interaction between these contributions appeared curial to solve hard software models synthesis benchmarks. Our implementation, called DFASAT, won the StaMinA competition.

Keywords software model synthesis, model inference, automaton identification, learning, satisfiability, state machines

1 Introduction

The behavior of software systems can often be specified using finite state machine models. These models provide an overview of software systems, by describing the way in which they react to different inputs, and when they produce which output. In addition, they allow for using techniques such as model checking (Clarke, 1997)

The first author is supported by the Austrian Science Foundation (FWF) NFN Grant S11408-N23 (RiSE). The second author is supported by STW project 11763 (ITALIA) and the Research Foundation - Flanders (FWO-Vlaanderen) project G.0682.11 (Declarative experimentation).

Johannes Kepler University, Linz, Austria, and Delft University of Technology, Delft, the Netherlands, E-mail: marijn@heule.nl · Radboud Universiteit Nijmegen, Nijmegen, the Netherlands, and Katholieke Universiteit Leuven, Leuven, Belgium, E-mail: sicco@cs.ru.nl

and model-based testing (Broy et al, 2005) to ensure that the software is operating correctly. Unfortunately, due to the time and cost involved in generating and maintaining these models, the construction of such models is often omitted during software development (Walkinshaw et al, 2010). An alternative to constructing these models manually, is to use *software model synthesis* (or process discovery) to derive them automatically from known software behavior (Cook and Wolf, 1998).

Usually, software synthesis tools model this behavior using deterministic finite state automata (DFAs). A DFA is a well-known language model that can be used to recognize a regular language, see, e.g., (Sudkamp, 2006). The synthesis relies on algorithms that can *identify* (or *infer* or *learn*) such a language model from a sample of labeled execution traces. This problem of identifying a DFA model from set of given examples is one of the best studied problems in grammatical inference, see, e.g., (de la Higuera, 2010). Identifying a DFA typically means finding the smallest DFA that is consistent with the set of examples. The size of a DFA is measured by the amount of states it contains. It is desired that this DFA is as small as possible because of an important principle known as Occam's razor, which states that among all possible explanations for a phenomenon, the simplest one is to be preferred. A smaller DFA is simpler, and therefore a better explanation and more likely model for the observed examples. DFA identification has also many applications in, for example, computational linguistics, bio-informatics, software verification, and speech processing, see (de la Higuera, 2010).

The problem of finding a smallest consistent DFA can be very hard. It is the optimization variant of the problem of finding a consistent DFA of a fixed size, which has been shown to be NP-complete (Gold, 1978). Even more troublesome is the result that the optimization version cannot be approximated (Pitt and Warmuth, 1989). In spite of these hardness results, quite a few DFA identification algorithms exist, see (de la Higuera, 2010). For many years, the state-of-the-art in DFA identification has been the *evidence-driven state-merging* (EDSM) algorithm (Lang et al, 1998). An evidence-driven algorithm is one that uses statistical evidence in order to determine which way to proceed. State-merging is a common technique from grammatical inference for learning a small language model by combining (merging) the states of a large initial model, see, e.g., (de la Higuera, 2010). Essentially, EDSM is a *greedy method* that tries to find a good local optimum efficiently. In addition, it has been shown using an earlier state-merging method called RPNI, that it *converges efficiently* (from polynomial time and data) to the global optimum in the limit (Oncina and Garcia, 1992). EDSM participated in and won in a tie with the search-intensive SAGE algorithm Juillé and Pollack (1998) the Abbadingo DFA learning competition in 1997 (Lang et al, 1998).

Since this competition, there have been few significant improvements in DFA identification from labeled examples. There has been some research into *specialized search procedures* for EDSM that typically lead to better results, see (Oliveira and Marques-Silva, 1998; Abela et al, 2004; Lang, 1999; Bugalho and Oliveira, 2005). These search techniques perform very well on small problems such as identifying a DFA with 20 states and an alphabet of size 2. On such problem instances, these search techniques often return the optimal solution within a few minutes, see for instance (Lang, 1999). Although the different search techniques improve the performance of EDSM, they are much less advanced than solvers for well-studied problems such as *graph coloring* and *satisfiability* (SAT). Especially SAT solvers have become very powerful in the last decade, using techniques such as conflict

analysis, intelligent back-jumping, and clause learning, see, e.g., (Biere et al, 2009). The power of such solvers can be used in other problems by *translating* these problems into SAT instances, and subsequently running a SAT solver on these translated problems. This approach is very competitive for several problems, see, e.g., (Biere et al, 1999; Marques-Silva and Glass, 1999; Endrullis et al, 2008). Recently, we successfully adopted this approach for DFA identification (Heule and Verwer, 2010).

In this paper, we apply our *SAT-based approach* to the problem of *software model synthesis*, i.e., learning the state machine model for the behavior of a given software system. This is a DFA identification problem. However, in contrast to most studies of DFA identification (see, e.g., (de la Higuera, 2010)), software models typically make use of a *large alphabet* of possible events. In addition, typically only a small portion of all possible behaviors will ever be seen in practice, leading to *sparse data sets*. The goal of the StaMinA DFA learning competition (Walkinshaw et al, 2010) was to find DFA learning techniques that perform well in this challenging setting. To this aim, the competition organizers generated data sets from 100 random DFAs that matched this setting. The state-of-the-art EDSM algorithm performs very poorly on these problems, achieving only 52% accuracy (only 2% better than random guessing) on the most difficult ones (a size 50 alphabet and observing only 12.5% of the input examples). With our SAT-based method and several additional techniques (a new heuristic, random greedy, and ensemble techniques), we improved the accuracy on these problems to 95%. Furthermore, we achieved the required 99% accuracy on all problem instances with an alphabet of size 50 and observing 50% of the input examples, which was significantly better than other approaches that participated in the StaMinA competition. These results show big potential for solving real-world software synthesis problems and constitute a significant improvement in state-of-the-art DFA identification.

Our final SAT-based method is a *unique combination of exact and greedy techniques* that works as follows. Initially, the StaMinA problem instances are too large for the SAT solver. By performing greedy EDSM steps, the size of the remaining problem becomes smaller and smaller. This continues until the problem is solvable using the SAT solver, at that time we switch to solving the remaining problem exactly. We believe that such a strategy is well-suited to machine learning methods in general, because these typically use heuristics that are based on statistics. Since every greedy step typically divides the data used to estimate these statistics, these statistics will at some point be estimated poorly. When this happens it makes sense to switch to an exact strategy.

In addition to providing this unique combination, we develop methods that solve two challenges faced when applying DFA identification to software model synthesis. First and foremost, the *bias of learning is different*. Instead of finding a smallest consistent DFA, the goal is to find a smallest consistent software model. This is a specific type of DFA in which many states can only generate a small subset out of a large set of possible events. Furthermore, only a few of these states generate exactly the same events. The overlap between generated events is therefore an important indicator for the similarity between states. We develop a *new heuristic for EDSM* aimed at identifying software models based on this overlap. In addition, since no heuristic is perfect, we employ the simple but surprisingly effective *random greedy* method to randomize the new heuristic values. The second challenge is that the *data is very sparse*. Because of this sparseness, it is unlikely

that our algorithm finds the DFA that was used to generate the data. To answer this challenge, we generate many good DFA software models using random greedy, and generalize over them using an *ensemble method* (Dietterich, 2000). Our final algorithm is the combination of each of these methods. We believe this combination of greedy and exact techniques to be crucial in solving the StaMinA problem instances (Walkinshaw et al, 2010), and an important step forward in DFA identification.

This paper is organized at follows. We start with an introduction to software model synthesis and the StaMinA competition (Section 2), followed by an overview of DFAs and the state-of-the-art in DFA identification (Section 3). We then provide a detailed explanation of our SAT-based approach (Section 4). Subsequently (Section 5), we explain the combination of greedy and exact methods, the new EDSM heuristic, the random greedy technique, and the used ensemble method. We give a complete overview of our final algorithm (Section 6), and describe the results obtained during the StaMinA competition (Section 7). We end this paper with some concluding remarks and ideas for future work (Section 8).

2 Software model synthesis and the StaMinA competition

The behavior of software systems can often be specified using finite state machine models. These models provide an overview of software systems, by describing the way in which they react to different inputs, and when they produce which output. Visualizing such machines can provide insights into the behavior of a software system, which can be of vital importance during the design and specification of such a system. Moreover, these machine allow for using automated techniques such as model checking (Clarke, 1997) and model-based testing (Broy et al, 2005) in order to ensure correctness of the system. For instance, these techniques can help to verify that the software system meets its requirements, or to test whether it will interact correctly with its environment. Unfortunately, due to the time and cost involved in generating and maintaining finite state machines, the construction of these models is often omitted during software development (Walkinshaw et al, 2010). An alternative to constructing these models manually, is to use *software model synthesis* (or process discovery) tools in order to derive them automatically from known software behavior (Cook and Wolf, 1998).

Software model synthesis (or system identification/learning, or process discovery/mining) is a technique for automatically constructing a software model based on observed system behavior. In software systems, this data typically consists of *execution traces*, i.e., sequences of operations, function calls, user interactions, or protocol primitives, which are produced by the system or its surrounding environment. Intuitively, software model synthesis tries to discover the logical structure (or model) underlying these sequences of events. This can be seen as a grammatical inference problem in which the events are modeled as the symbols of a language, and the goal is to find a model for this language. Many different language models and ways of finding them are available in the grammatical inference, machine learning, and data mining literature. Which one to choose depends mostly on the available data and the type of system under consideration.

In this paper, we assume the availability of *labeled data* (or positive and negative examples). This means that both desired (positive) and undesired (negative)

execution traces are available. Other common identification settings are identification from unlabeled data (only positive examples) and identification from queries (or query-learning), see, e.g., (Kearns and Vazirani, 1994). Identification from unlabeled data is mostly the domain of probabilistic language models and identification algorithms that are based on statistics, e.g., (Clark and Thollard, 2004). Identification from unlabeled data is less powerful than learning from labeled data and often a lot more data is required in order to obtain good performance of the identification algorithms, see (Jain et al, 1999). Fortunately, this type of data is often very easy to obtain: simply observe the events generated by a system. In contrast, labeled data requires some labeling to process (often performed by a domain expert) to determine the ones that are desired and the ones that are not. In identification from queries, access to an oracle is needed that can answer specific types of questions such as: whether a specific string is part of the language (membership queries), and whether a given model is a model for the language (equivalence queries). In software model synthesis, the actual software system can be used for this purpose, see, e.g., (Raffelt et al, 2009). When such an oracle is available that can be queried often and quickly, it is advisable to use it as much as possible since identification from queries is very powerful, making it possible to identify very large realistic models. Due to time constraints, however, there often is a limit to the amount of available queries. In such cases, it is also possible to combine identification from queries and data, see, e.g., (Dupont et al, 2008).

The model we use in this paper is the deterministic finite state automaton (DFA), see, e.g., (Sudkamp, 2006). This model is very popular for specifying the behavior of software systems. Furthermore, identifying a DFA is one of the best studied problems in grammatical inference, and many algorithms have been developed for this purpose (de la Higuera, 2010). A DFA is a simple model, however, and in some cases it will not be able to represent or identify all the complex behaviors of a software system. Some more powerful models with identification algorithms include: non-deterministic automata (Yokomori, 1993; Denis et al, 2000), probabilistic automata (Clark and Thollard, 2004; Castro and Gavaldà, 2008), Petri-nets (modeling concurrency) (van der Aalst, 2011), timed automata (Verwer, 2010; Grinchtein et al, 2006a), I/O automata (modeling both input and output) (Aarts and Vaandrager, 2010), and Büchi automata (modeling infinite strings) (Higuera and Janodet, 2004). Despite its limited power, DFA learning methods have been used to learn different types of complex systems such as web-services (Bertolino et al, 2009), X11 windowing programs (Ammons et al, 2002), network protocols (Cui et al, 2007; Antunes et al, 2011), and java programs (Walkinshaw et al, 2007; Dallmeier et al, 2006; Mariani et al, 2011). Unfortunately, nearly all of the comparative studies of DFA identification methods (e.g, (Lang et al, 1998; Bugalho and Oliveira, 2005)) cover settings that are very different from the ones in software engineering. In particular, the studied models usually contain a small alphabet and are identified from very large and complete sets of event sequences. In software models, however, typically make use of a large set of possible events. In addition, typically only a small portion of all possible behaviors will ever be seen in practice, leading to sparse data sets (Walkinshaw et al, 2010).

The goal of the StaMinA DFA learning competition (Walkinshaw et al, 2010) was to find DFA learning techniques that perform well in this challenging setting. To this aim, the competition organizers generated labeled data sets from random DFAs that matched with this setting. The difficulty of the different problem in-

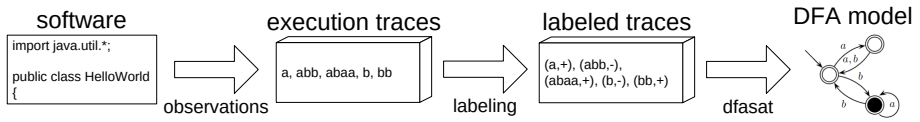


Fig. 1 Based on observed execution traces of a software system, labeled as desired and undesired, the dfasat algorithm can be used to find DFA models for the desired behavior of the software system.

stances was varied using differently sized alphabets and different levels of sparsity. For alphabets of size 2, 5, 10, 20, and 50, different DFAs were generated which all contained about 50 states. From these DFAs large sets of labeled training instances (in total 20,000) were sampled using a random walk procedure. Sparsity was then introduced by resampling the training data provided to the participants. The resampled sets contained either 100%, 50%, 25%, or 12.5% of the original training data. For every combination of alphabet size and sparsity level, 5 different DFAs and labeled sets were generated. In order to measure the performance of the different algorithms, participants were asked to provide the labels of a small set (1,500 instances) of unseen test data. These were then compared to the actual labels provided by the generated DFA using a balanced classification rate (BCR, a variant of accuracy), see Section 7. When a participant managed to achieve an accuracy of 99% on all 5 problems of a certain size-sparsity combination, (s)he was said to "break" the corresponding cell in the StaMinA problem grid (see `http://stamina.chefbe.net`). The participant that managed to break the most difficult cell the earliest was announced as the winner.

The state-of-the-art DFA identification algorithm (evidence-driven state-merging (Lang et al, 1998)) performs very poorly on the StaMinA problems, achieving only 52% accuracy on the most difficult ones (a size 50 alphabet and observing only 12.5% of the input examples). This is an improvement of only 2% over random guessing. In this paper, we present the winner of the StaMinA competition: dfasat. Our algorithm improved the accuracy on these problems to 95%. Furthermore, we achieved the required 99% accuracy on all problem instances with an alphabet of size 50 and observing 50% of the input examples, which was significantly better than other approaches that participated in the StaMinA competition. These results show big potential for solving real-world software synthesis problems and constitute a significant improvement in state-of-the-art DFA identification.

The dfasat algorithm can be used to identify DFA models for software systems based on labeled data, see Figure 1. On its own, this can provide important insights into the inner workings of a software system and its environment. In combination with other techniques, however, it can become a very powerful method for software engineering tasks such as modeling, requirement engineering, maintenance, verification, and testing.

3 The state-of-the-art in DFA identification

A *deterministic finite state automaton* (DFA) is one of the basic and most commonly used finite state machines. Below, we provide a concise description of DFAs,

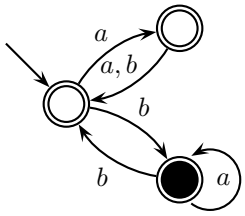


Fig. 2 A deterministic finite state automaton (DFA). It contains states (circles) and labeled transitions (arrows). One state is a predefined start state, indicated by the arrow pointing from nowhere. Accepting states have a white center, rejecting states have a black center. Strings that end in accepting states are accepted, others rejected. For example, the DFA accepts *a*, *abaa*, and *bb*, and rejects *b*, and *abb*.

the reader is referred to (Sudkamp, 2006) for a more elaborate overview. A DFA $A = \langle Q, T, \Sigma, q_0, Q_+ \rangle$ is a directed graph consisting of a set of *states* Q (nodes) and labeled *transitions* T (directed edges). An example is shown in Figure 2. The *start state* $q_0 \in Q$ is a specific state of the DFA and any state can be an *accepting state* (final state) in $Q_+ \subseteq Q$. The labels of transitions are all members of a given *alphabet* Σ. A DFA A can be used to *generate* or *accept* sequences of symbols (strings) using a process called *DFA computation*. This process begins in q_0, and iteratively *activates* (or *fires*) an outgoing transition $t_i = \langle q_{i-1}, q_i, l_i \rangle \in T$ with label $l_i \in \Sigma$ from the *source state* it is in q_{i-1}, moving the process to the *target state* q_i pointed to by t_i. A computation $q_0 t_1 q_1 t_2 q_2 \ldots t_n q_n$ is *accepting* if the state it *ends* in (its last state) is an accepting state $q_n \in Q_+$, otherwise it is *rejecting*. The labels of the activated transitions form a string $l_1 \ldots l_n$. A DFA accepts exactly those strings formed by the labels of accepting computations, it rejects all others. Since a DFA is *deterministic* there exists exactly one computation for every string, implying that for every state q and every label l there exists at most one outgoing transition from q with label l. A string s is said to *reach* all the states contained in the computation that forms s, s is said to *end* in the last state q_n of such a computation. The set of all strings accepted by a DFA A is called the *language* $L(A)$ of A.

Given a pair of finite sets of positive example strings S_+ and negative example strings S_-, called the *input sample*, the goal of *DFA identification* (or *learning*) is to find a (non-unique) *smallest* DFA A that is *consistent* with $S = \{S_+, S_-\}$, i.e., such that every string in S_+ is accepted by A, and every string in S_- is rejected by A. Typically, the size of a DFA is measured by the number of states it contains. Seeking this DFA is an active research topic in the grammatical inference community, see, e.g., (de la Higuera, 2010). The performance of a DFA identification algorithm is typically measured using another set of positive and negative examples, called the *test sample*. Both the input sample and the test sample come from the same DFA language. In other words, there exists a (preferably small) DFA A_t that is consistent with both the input sample and the test sample. This DFA A_t, called the *target DFA*, is assumed to have generated (or really did generate) the input sample. Typically, the *accuracy* of the identified DFA A on the test sample is used as a performance measure. Accuracy is the number of correctly classified

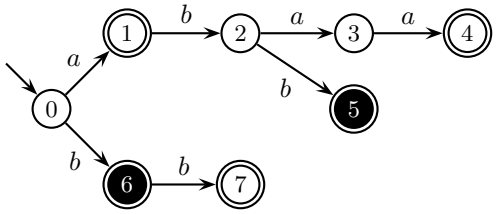


Fig. 3 An augmented prefix tree acceptor for $S = (S_+ = \{a, abaa, bb\}, S_- = \{abb, b\})$. The start state is the state with an arrow pointing to it from nowhere.

(accepted or rejected) examples divided by the total amount of examples. This value is determined by checking whether the computation of A on every example in the test sample corresponds to its label (positive or negative). In StaMinA, this measure was balanced between the positive and negative examples, see Section 7. Intuitively, accuracy tests how good the identification algorithm is at finding the target DFA A_t that generated the input sample. The current state-of-the-art in DFA identification is evidence-driven state-merging in the red-blue framework (EDSM) (Lang et al, 1998), possibly with some search procedure wrapped around it in order to continue searching once a possible local optimum has been reached. In the following, we explain this algorithm and the used search techniques.

3.1 State Merging

The idea of a state-merging algorithm is to first construct a tree-shaped DFA A from the input sample S (Algorithm 1), and then to merge the states of A (Algorithm 2). This DFA A is called an *augmented prefix tree acceptor* (APTA). An example is shown in Figure 3. For every state q of A, there exists exactly one computation that ends in q. This implies that the computations of two strings s and s' reach the same state q if and only if s and s' share the same prefix until they reach q. Furthermore, an APTA A is constructed to be *consistent* with the

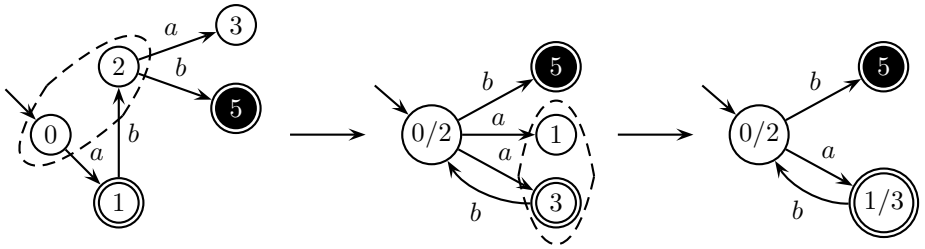


Fig. 4 A merge of two states: $0, 2$ from the APTA of Figure 3. On the left the original part of the automaton is shown. The states that are to be merged are surrounded by a dashed ellipse. In the middle the result of that merge is shown. The resulting automaton still has to be determinized by merging the children $1, 3$ that can be reached using transitions with the same label from $0/2$. On the right the result after determinization is shown.

Algorithm 1 Construct the APTA: APTA

Require: an input sample $S = \{S_+, S_-\}$
Ensure: A is the APTA for S
  $\Sigma = \bigcup_{s \in S} \{l \mid l \in s\}$ // Σ is the set of all symbols in S
  $A = \langle Q = \{q_0\}, T = \emptyset, \Sigma, q_0, Q_+ = \emptyset, Q_- = \emptyset \rangle$ // A is an empty DFA
  for all $s = l_1, \ldots, l_n \in S$ **do** // for every input string s
    state $q = q_0$ // let q be the start state
    for $0 \leq i \leq n$ **do** // iterate over the labels l_i in s
      if there exists no $\langle q, q', l_i \rangle \in T$ **then** // if there is no transition from q with label l_i
        create q', set $Q := Q \cup \{q'\}$ // add a new state q' to A
        set $T := T \cup \{\langle q, q', l_i \rangle\}$ // create a transition to q' from q with label l_i
      end if
      $q = q'$ such that $\langle q, q', l_i \rangle \in T$ // update q by following the transition with label l_i
    end for
    if $s \in S_+$ **then**
      $Q_+ = Q_+ \cup \{q\}$ // make the state s ends in accepting if s is positive
    else
      $Q_- = Q_- \cup \{q\}$ // make it rejecting otherwise
    end if
  end for
  return A

input sample S, i.e., $S_+ \subseteq L(A)$ and $S_- \cap L(A) = \emptyset$. Thus a state q is accepting only if there exists a string $s \in S_+$ such that the computation of s ends in q. Similarly, it is rejecting only if the computation of a string $s \in S_-$ ends in q. As a consequence, A can contain states that are neither accepting nor rejecting. None of the computations of strings from S ends in such a state. Therefore, the rejecting states are maintained in a separate set $Q_- \subseteq Q$, with $Q_- \cap Q_+ = \emptyset$. Whether a state $q \in Q \setminus (Q_+ \cup Q_-)$ should be accepting or rejecting is determined by merging the states of the APTA and trying to find a DFA that is as small as possible.

A *merge* (see Figure 4 and Algorithm 2) of two states q and q' combines the states into one: it creates a new state q'' that has the incoming and outgoing transitions of both q and q', i.e., replace all $\langle q, q_t, l \rangle, \langle q', q_t, l \rangle \in T$ by $\langle q'', q_t, l \rangle$ and all $\langle q_s, q, l \rangle, \langle q_s, q', l \rangle \in T$ by $\langle q_s, q'', l \rangle$ (the for loops in Algorithm 2). Such a merge is only allowed if the states are *consistent*, i.e., it is not the case that q is accepting while q' is rejecting or vice versa. When a merge introduces a non-deterministic choice, i.e., q'' is now the source of two transitions $\langle q'', q_1, l \rangle$ and $\langle q'', q_2, l \rangle$ in T with the same label l, the target states of these transitions q_1 and q_2 are merged as well. This is called the *determinization* process (the while loop in Algorithm 2), and is continued until there are no non-deterministic choices left. However, if this process at some point merges two inconsistent states, the original states q and q' are also considered inconsistent and the merge will fail. The result of a successful merge is a new DFA that is smaller than before, and still consistent with the input sample S. A state-merging algorithm iteratively applies this state merging process until no more consistent merges are possible. Notice that when states q_0, q_2 and q_7 of the APTA of Figure 3 are merged, then a part of the DFA of Figure 2 is obtained.

The successful *red-blue framework* (Lang et al, 1998) follows the state-merging algorithm just described, and in addition adds colors (red and blue) to the states to guide the merge process. A red-blue algorithm only merges red $r \in R \subseteq Q$ and blue $b \in B \subseteq Q$ states. The red states and the transitions between them form the

Algorithm 2 Merging two states: merge $(A,\ q,\ q')$

Require: an augmented DFA $A = \langle Q, T, \Sigma, q_0, Q_+, Q_- \rangle$ and two states $q, q' \in Q$
Ensure: if q and q' are inconsistent, return FALSE; else return A with q and q' merged.

```
if (q ∈ Q_+ and q' ∈ Q_-) or (q ∈ Q_- and q' ∈ Q_+) then
    return FALSE                                   // return FALSE if q is inconsistenct with q'
end if
let A' ⟨Q', T', Σ, q'_0, Q'_+, Q'_-⟩ be a copy of A          // initialize the result A'
create a new state q'', and set Q' := Q' ∪ q''                // add a new state q'' to A'
if q ∈ Q_+ or q' ∈ Q_+ then
    set Q'_+ := Q'_+ ∪ {q''}                       // q'' is accepting if q or q' is accepting
end if
if q ∈ Q_- or q' ∈ Q_- then
    set Q'_- := Q'_- ∪ {q''}                       // q'' is rejecting if q or q' is rejecting
end if
for all t = ⟨q_s, q_t, l⟩ ∈ T' with q_s ∈ {q, q'} do // forall transitions with source state q or q'
    T' := T' \ { t}                                                    // remove the transition
    T' := T' ∪ {⟨q'', q_t, l⟩}                     // add a new transition with q'' as source
end for
for all t = ⟨q_s, q_t, l⟩ ∈ T' with q_t ∈ {q, q'} do // forall transitions with target state q or q'
    T' := T' \ { t}                                                    // remove the transition
    T' := T' ∪ {⟨q_s, q'', l⟩}                     // add a new transition with q'' as target
end for
set Q' := Q' \ {q, q'}                                          // remove q and q' from A'
while ⟨q_f, q_1, l⟩, ⟨q_f, q_2, l⟩ ∈ T' with q_1 ≠ q_2 do // while non-deterministic choices exist
    A'' := merge(A', q_1, q_2)                                  // determinize the targets
    if A'' equals FALSE then
        return FALSE                               // return FALSE if the targets are inconsistent
    else
        A' := A''                                  // else keep the merge and continue determinizing
    end if
end while
return A'
```

currently constructed DFA, the blue states are still to be identified transitions, potentially to new states of the DFA. The new state q'' resulting from a red-blue merge is colored red, i.e., $R := R \cup \{q''\}$. In addition, every non-red target state $q \in Q \setminus R$ that is the target of a transition $\langle r, q, l \rangle \in T$ with a red source state $r \in R$, is colored blue, i.e., $B := B \cup \{q\}$. In this way, the framework maintains a core of red states with a fringe of blue states (see Figure 5 and Algorithm 3). Initially, the start state of the APTA is colored red, and its children (targets for every symbol) are colored blue.

Note that every blue state is the root of a tree of uncolored states. Consequently, every pair of states q and q' that is merged by the determinization process contains at most one colored state. The state q'' resulting from a determinization merge of q or q' is given this color. A determinization merge of two uncolored states leaves q'' uncolored. Whenever there exists a blue state b for which no consistent merge is possible with a red state ($\mathsf{merge}(A, b, r)$ is FALSE for all $r \in R$), the algorithm changes the color of this blue state into red ($R := R \cup \{b\}$). Since a red-blue state-merging algorithm never merges pairs of red states, it is guaranteed not to modify the transitions between red states. In other words, once $\langle r, r', l \rangle \in T$ with $r, r' \in R$ has been created in an iteration of a red-blue state-merging algorithm, $\langle r, r', l \rangle$ will be in the DFA returned by the algorithm. The red core of the DFA

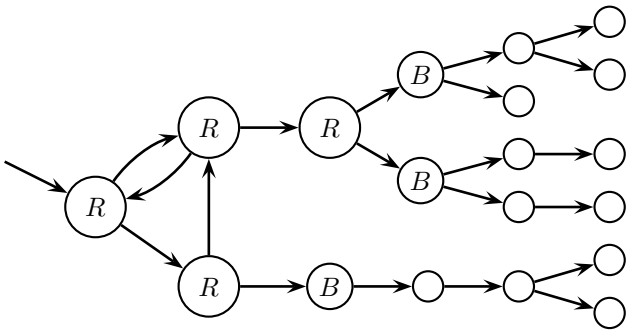


Fig. 5 The red-blue framework. The red states (labeled R) are the identified parts of the automaton. The blue states (labeled B) are the current candidates for merging. The uncolored states are pieces of the APTA. The symbols on the transitions are omitted for clarity.

can be thus be viewed as a part of the DFA that is (assumed to be) correctly identified.

A red-blue state-merging algorithm is complete since it is capable of producing any DFA that is consistent with the input sample and smaller than the original APTA. Furthermore, it is *more efficient* (in terms of computation time) than standard state-merging since it considers a lot less merges. Currently, the most successful method for solving the DFA identification problem is the *evidence driven state-merging* (EDSM) algorithm in the red-blue framework (Lang et al, 1998). In EDSM each possible merge is given a score based on the amount of *evidence* in the merges that are performed by the merge and determinization processes. Let $|A|$ denote the sum of the number of accepting and rejecting states in A, i.e., $|A| = |Q_+ \cup Q_-|$. The evidence score is equal to:

$$\mathsf{evidence}(A, q, q') := \begin{cases} |A| - |\mathsf{merge}(A, q, q')| & \text{if } \mathsf{merge}(A, q, q') \neq \text{FALSE} \\ -1 & \text{otherwise} \end{cases}$$

In other words, the difference in accepting and rejecting states before and after performing the merge. In every iteration, EDSM chooses the merge that maximizes this difference, which is computed by counting the number of accepting states that are merged with accepting states and adding the number of rejecting states that are merged with rejecting states. EDSM can be seen as a greedy procedure that uses this simple heuristic to determine which merge to perform. Intuitively, this heuristic tries merge the states that have the most statistical evidence (confidence) of being the a single state in the DFA that generated the data. Under the assumption that every merge during the determinization process is an independent statistical test that has some probability to show that the merge is inconsistent, the heuristic chooses the merge that passes the most of these tests. EDSM participated in and won (in a tie) the Abbadingo DFA learning competition in 1997 (Lang et al, 1998). In the competition EDSM was capable of approximately (with 99%

Algorithm 3 State-merging in the red-blue framework

Require: an input sample S
Ensure: A is a DFA that is consistent with S
  $A = \mathsf{apta}(S)$ // construct the APTA A
  $R = \{q_0\}$ // color the start state of A red
  $B = \{q \in Q \setminus R \mid \exists \langle q_0, q, l\rangle \in T\}$ // color all its children blue
  while $B \neq \emptyset$ **do** // while A contains blue states
    if $\exists b \in B$ s.t. $\forall r \in R$ holds $merge(A, r, b) = \text{FALSE}$ **then** // if there a blue state inconsistent with every red states
      $R := R \cup \{b\}$ // color b red
      $B := B \cup \{q \in Q \setminus R \mid \exists \langle b, q, l\rangle \in T\}$ // color all its children blue
    else
      for all $b \in B$ and $r \in R$ **do** // forall red-blue pair of states
        compute the $\mathsf{evidence}(A, q, q')$ of $merge(A, r, b)$ // find the best performing merge
      end for
      $A := merge(A, r, b)$ with highest $\mathsf{evidence}$ // perform the best merge
      let q'' be resulting state
      $R := R \cup \{q''\}$ // color the resulting state red
      $R := R \setminus \{r\}$ // uncolor the merged red state
      $B := \{q \in Q \setminus R \mid \exists r \in R \text{ and } \langle r, q, l\rangle \in T\}$ // recompute the set of blue states
    end if
  end while
  return A

accuracy) learning a DFA with 500 states with a training set consisting of $60,000$ strings on a 2 letter alphabet.

In the grammatical inference community, there has been some research into developing advanced and efficient search techniques for ESDM. The idea is to increase the quality of a solution by searching other paths in addition to the path determined by the greedy EDSM heuristic. Examples of such advanced techniques are dependency directed backtracking (Oliveira and Marques-Silva, 1998), using mutually (in)compatible merges (Abela et al, 2004), and searching most-constrained nodes first (Lang, 1999). A comparison of different search techniques for EDSM can be found in (Bugalho and Oliveira, 2005).

The current state-of-the-art techniques are two simple search strategies called ed-beam and exbar (Lang, 1999). The ed-beam procedure calculates one greedy EDSM path starting from every node in the search tree in breadth-first order. In other words, it tries all possible merges (from a given search node), adds the resulting search nodes to a queue, computes the DFA result of an EDSM run starting from these nodes, pops a new search node from the queue, and iterates. The smallest DFA found by these EDSM runs is returned as a solution. This solution then serves as an upper bound of the DFA size for the breadth-first search. The exbar procedure iteratively runs a full search of the EDSM search space with an increasing upper bound on the number of DFA states. When it reaches this bound it backtracks the last heuristic decision, and if all search nodes with up to the bound have been searched, it increases the bound and iterates. It continues this procedure until a solution is found. In addition, in order to reduce the size of the search space, exbar searches the most-constrained nodes first. Typically, a time bound is set and the algorithm is stopped when its running-time exceeds this bound. However, it can guarantee that it has found an optimal solution (a smallest DFA) if all smaller solutions have been visited by its search procedure.

4 SAT-based DFA identification

Recently, instead of wrapping a search technique around EDSM, we proposed to *translate* the DFA identification problem into satisfiability (SAT) and then use a state-of-the-art SAT-solver to search for an optimal solution (Heule and Verwer, 2010). The main advantage of such an approach is that it directly makes use of advanced search techniques such as conflict analysis, intelligent back-jumping, and clause learning, see, e.g., (Biere et al, 2009). Despite the low level representation, such an approach is very competitive for several problems. Examples are bounded model checking (Biere et al, 1999), equivalence checking (Marques-Silva and Glass, 1999) and rewriting termination problems (Endrullis et al, 2008). In addition, a nice bonus is that due to the yearly SAT competition, the performance of these solvers improves every year due to the hard labor of fellow researchers.

Our method is inspired by the translation in (Coste and Nicolas, 1997) from DFA identification into *graph coloring*. Graph coloring is the problem of assigning a color to every node in a given graph such that nodes with the same color do not share an edge. Determining whether there exists a coloring that uses at most $k \geq 3$ colors is a well-known NP-complete problem, see, e.g., (Garey and Johnson, 1979). The main idea of the translation into graph coloring is to use a distinct color for every state of the identified DFA. The nodes in the graph coloring instance represent the labeled examples and share an edge if one of them is positive and the other negative. The graph coloring problem thus ensures that inconsistent examples cannot obtain the same color, and therefore cannot end in the same state, making the resulting DFA consistent. The size of this DFA is determined by the amount of colors used in the graph coloring problem. Finding the minimum can be done by iterating over this amount.

The *satisfiability* problem (SAT) (Biere et al, 2009) deals with the question whether there exists an assignment to Boolean variables such that a given formula in conjunctive normal form (CNF) evaluates to true. Such a formula is a conjunction ($\wedge$) of clauses, each clause being a disjunction ($\vee$) of literals. Literals refer either to a Boolean variable x or to its negation $\neg x$. One of the main problems we solved with our method is how to efficiently encode the graph coloring constraints of (Coste and Nicolas, 1997) into SAT. A naive *direct encoding* (Walsh, 2000) of these constraints would lead to $O(k^2|V|^2)$ clauses, where k is the size of the identified DFA, and $|V|$ is the size (number of states) of the APTA for the labeled examples. Since this APTA combines prefixes of these examples, not only the amount but also the length of these examples has an effect on $|V|$. Our encoding requires only $O(k^2|V|)$ clauses. A closely related approach (Grinchtein et al, 2006b) uses a translation of DFA identification into an integer constraint satisfaction problem (CSP) from (Biermann and Feldman, 1972). It then translates this CSP into SAT in two ways: using a unary and a binary encoding of the integers. Interestingly, a direct encoding of graph coloring is identical to a unary encoding of the CSP constraints. Again, the minimum is found by iterating over the number of states.

The crucial part of our translation is the use of *auxiliary variables* to represent the problem more efficiently. In addition, we apply *symmetry breaking* (Sakallah, 2009) to prevent overlapping searches with different colors by preprocessing the result of our translation with a fast max-clique approximation algorithm. Furthermore, we add many *redundant clauses* to our translation that provide the SAT

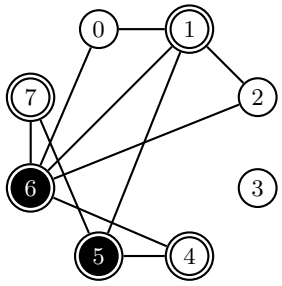


Fig. 6 The consistency graph corresponding to the APTA of Figure 3. Some states in the consistency graph are not directly inconsistent, but inconsistent due to determinization. For instance states 2 and 6 are inconsistent because the strings *abb* (negative) and *bb* (positive) would end in the same state if these states were merged. Similarly, states 1 and 2 are inconsistent because the strings *a* (positive) and *abb* (negative) would end in the same state if these states were merged.

solver with some additional knowledge about the DFA identification instance. In the following, we describe each of these techniques in turn.

4.1 From DFA identification to graph coloring

The main idea of the translation in (Coste and Nicolas, 1997) is to use a distinct color for every state of the identified DFA. Every node in the graph coloring problem corresponds to a distinct state in the APTA. With slight abuse of notation, we refer to these nodes using the corresponding APTA states. Two states v and w in this graph are connected by an edge (cannot be assigned the same color), if merging v and w results in an inconsistency (i.e., an accepting state is merged with a rejecting state). These edges are called *inequality constraints*. Figure 6 shows an example of such a graph.

In addition to these inequality constraints, *equality constraints* are required: if the parents $p(v)$ and $p(w)$ of two states v and w (in the APTA) with the same incoming transition label are merged, then v and w must be merged too. The incoming transition label of an APTA state v is the label of the unique transition that has v as its target. In the graph coloring problem, these equality constraints imply that the two parent states $p(v)$ and $p(w)$ can get the same color only if v and w get the same color. Such a constraint is difficult to implement in graph coloring. In (Coste and Nicolas, 1997), this is dealt with by modifying the graph according to the consequences of these constraints. This implies that a new graph coloring instance has to be solved every time an equality constraint is used. In our SAT-based method we encode these constraints directly into SAT.

4.2 Direct encoding

A widely used translation of graph coloring problems into SAT is known as the *direct encoding* (Walsh, 2000). Given a graph $G = (V, E)$ and a set of colors $C = \{1, \ldots, k\}$, the direct encoding uses Boolean *color variables* $x_{v,i}$ with $v \in V$

and $i \in C$. If $x_{v,i}$ is assigned to true, it means that state (vertex) v has color i. The constraints on these variables are as follows (see Table 1 for details): For every state, *at-least-one* color clauses ensure that each state is colored, while *at-most-one* color clauses forbid that a state can have multiple colors. The latter clauses are redundant because they are not required for a solution to be correct. Additionally, we have to translate that adjacent states cannot have the same color. The direct encoding uses the following clauses:

$$\bigwedge_{i \in C} \bigwedge_{(v,w) \in E} (\neg x_{v,i} \vee \neg x_{w,i})$$

Let EL be the set consisting of pairs of states that have the same incoming label in the APTA. In case the parents $p(v)$ and $p(w)$ of such a pair $(v, w) \in EL$ have the same color, then v and w must have the same color as well. This corresponds to the equality constraints in (Coste and Nicolas, 1997). A straight-forward translation of these constraints into CNF is:

$$\bigwedge_{i \in C} \bigwedge_{j \in C} \bigwedge_{(v,w) \in EL} \begin{array}{l} (\neg x_{p(v),i} \vee \neg x_{p(w),i} \vee \neg x_{v,j} \vee x_{w,j}) \wedge \\ (\neg x_{p(v),i} \vee \neg x_{p(w),i} \vee x_{v,j} \vee \neg x_{w,j}) \end{array}$$

This encoding is identical to the CSP-based translation given in (Grinchtein et al, 2006b). Notice that the size of the direct encoding is $O(k^2|V|^2)$. For interesting DFA identification problems this will result in a formula that will be too large for the current state-of-the-art SAT solvers. Therefore we will propose a more compact encoding below.

4.3 Compact encoding with auxiliary variables

The majority of clauses in the direct encoding originate from translating the equality constraints into SAT. We propose a more efficient encoding based on auxiliary variables $y_{a,i,j}$, which we refer to as *parent relation variables*. If set to true, $y_{a,i,j}$ means that for any state with color i, the child reached by label a has color j. Let $l(v)$ denote the incoming label of state v. As soon as both a child v and its parent $p(v)$ are colored, we force the corresponding parent relation variable to true by the clauses

$$\bigwedge_{i \in C} \bigwedge_{j \in C} \bigwedge_{v \in V} (y_{l(v),i,j} \vee \neg x_{p(v),i} \vee \neg x_{v,j})$$

This leads to $O(k^2|V|)$ clauses with only $k^2|\Sigma|$ additional literals (where $|\Sigma|$ is the size of the alphabet). Additionally, we require *at-most-one* parent relation clauses to guarantee that each relation is unique:

$$\bigwedge_{a \in \Sigma} \bigwedge_{h \in C} \bigwedge_{i \in C} \bigwedge_{j \in C, j > h} (\neg y_{a,i,h} \vee \neg y_{a,i,j})$$

This new encoding reduces the number of clauses significantly. To further reduce this size, we introduce an additional set of auxiliary variables z_i with $i \in C$. If z_i is true, color i is only used for accepting states. Therefore, we refer to them as *accepting color variables*. They are used for the constraint that requires accepting

states V_+ to be colored differently from rejecting states V_-. Without auxiliary variables, this can be encoded as

$$\bigwedge_{v \in V_+} \bigwedge_{w \in V_-} \bigwedge_{i \in C} (\neg x_{v,i} \vee \neg x_{w,i})$$

resulting in $|V_+| \cdot |V_-| \cdot k$ clauses. Using the auxiliary variables z_i, the same constraints can be encoded as

$$\bigwedge_{v \in V_+} \bigwedge_{i \in C} (\neg x_{v,i} \vee z_i) \wedge \bigwedge_{w \in V_-} \bigwedge_{i \in C} (\neg x_{w,i} \vee \neg z_i)$$

requiring only $(|V_+| + |V_-|)k$ clauses and k additional literals.

Table 1 Encoding of DFA identification into SAT. C = set of colors, Σ = set of labels (alphabet), V = set of states, E = set of conflict edges.

Variables	**Range**	**Meaning**
$x_{v,i}$	$v \in V; i \in C$	$x_{v,i} \equiv 1$ iff state v has color i
$y_{a,i,j}$	$a \in \Sigma; i,j \in C$	$y_{a,i,j} \equiv 1$ iff parents of states with color j and incoming label a have color i
z_i	$i \in C$	$z_i \equiv 1$ iff an accepting state has color i

Clauses	**Range**	**Meaning**
$(x_{v,1} \vee x_{v,2} \vee \cdots \vee x_{v,k})$	$v \in V$	every state has at least one color
$(\neg x_{v,i} \vee z_i) \wedge (\neg x_{w,i} \vee \neg z_i)$	$v \in V_+; w \in V_-; i \in C$	accepting states cannot have the same color as rejecting states
$(y_{l(v),i,j} \vee \neg x_{p(v),i} \vee \neg x_{v,j})$	$v \in V; i,j \in C$	a parent relation is set when a state and its parent are colored
$(\neg y_{a,i,h} \vee \neg y_{a,i,j})$	$a \in \Sigma; h,i,j \in C; h < j$	each parent relation can target at most one color
Redundant Clauses	**Range**	**Meaning**
$(\neg x_{v,i} \vee \neg x_{v,j})$	$v \in V; i,j \in C; i < j$	every state has at most one color
$(y_{a,i,1} \vee y_{a,i,2} \vee \cdots \vee y_{a,i,k})$	$a \in \Sigma; i \in C$	each parent relation must target at least one color
$(\neg y_{l(v),i,j} \vee \neg x_{p(v),i} \vee x_{v,j})$	$v \in V; i,j \in C$	a parent relation forces a state once the parent is colored
$(\neg x_{v,i} \vee \neg x_{w,i})$	$i \in C; (v,w) \in E$	all determinization conflicts explicitly added as clauses

4.4 Symmetry breaking

In case a graph cannot be colored with k colors, the corresponding (unsatisfiable) SAT instance will solve the problem $k!$ times: once for each permutation of the colors. Therefore, when dealing with CNF formulas representing graph coloring problems, it is good practice to add *symmetry breaking predicates* (SBPs) (Sakallah, 2009). Notice that in any valid coloring of the consistency graph, every state in a clique must have a different color. So, one can fix the color of states in a large

clique in this graph in a pre-processing step. Although finding the largest clique in a graph is NP-complete, a large clique K can be computed cheaply using a greedy algorithm:

1. Start with a state v with highest degree. Set $K = \{v\}$.
2. Let G_K be the subgraph of states that are connected to all states in K.
3. Add a state with highest degree in G_K to K.
4. Repeat from Step 2 until G_K contains only states in K.

Since every state in K needs to have a different color, and the actual value of the color does not matter for the solution, we simply pre-assign a color to every state in $K = \{v_1, \ldots, v_n\}$ using unit clauses:

$$\bigwedge_{v_i \in K} (x_{v_i,c(i)})$$

where $c(i)$ is the ith color.

4.5 Adding redundant clauses

The compact encoding discussed above can be extended with several types of redundant clauses. First, we can explicitly state that every state must be colored with exactly one color by adding the redundant *at-most-one color clauses*

$$\bigwedge_{v \in V} \bigwedge_{i \in C} \bigwedge_{j \in C, j > i} (\neg x_{v,i} \vee \neg x_{v,j})$$

Similarly, we can explicitly encode that for each combination of a color and a label exactly one parent relation variable must be true. This is achieved by adding the *at-least-one parent relation clauses*

$$\bigwedge_{a \in \Sigma} \bigwedge_{i \in C} (\bigvee_{j \in C} y_{a,i,j})$$

These two types of clauses are known as *blocked clauses* (Kullmann, 1999). These clauses have at least one literal that cannot be removed by resolution. Therefore, blocked clauses cannot be used to derive the empty clause (i.e., show that the formula is unsatisfiable) (Kullmann, 1999). So, formulas with and without blocked clauses are equisatisfiable. In previous work (Heule and Verwer, 2010), we showed that these blocked clauses improve the performance of DFA identification. For other problems, removal of blocked clauses results in a speed-up (Jarvisalo et al, 2010). Other types of redundant clauses we use include a reformulation of the constraint on the parent relation:

$$\bigwedge_{v \in V} \bigwedge_{i,j \in C} (\neg y_{l(v),i,j} \vee \neg x_{p(v),i} \vee x_{v,j})$$

and an explicitly formulation of all edges of the consistency graph that are not covered by the accepting color literals, i.e., the inconsistencies caused by determinization:

$$\bigwedge_{i \in C} \bigwedge_{(v,w) \in E} (\neg x_{v,i} \vee \neg x_{w,i})$$

Although these clauses are redundant, they provide some additional knowledge about the problem to the SAT solver. In particular, adding the clauses that represent inconsistent merges appeared to improve the performance in our previous work (Heule and Verwer, 2010). For the StaMinA problems, however, we excluded them since they only resulted in a large overhead (in the worst case $O(k|V|^2)$ clauses are required) and an increased solving time.

5 DFA identification for software model synthesis

We now describe how we applied our SAT-based approach to the problem of software model synthesis. The size of the alphabet, sparsity of the available data, and types of machines used to generate the data in the StaMinA problem instances (Walkinshaw et al, 2010) lead to several problems for our SAT-translation:

1. *The resulting encoding is too large for state-of-the-art SAT solvers.* Naively applying our SAT-based technique by iterating over the number of states quickly requires hundreds of millions of clauses.
2. *The learning bias (finding the smallest DFA) is unsuited for software model synthesis.* For example, in a software model, many states occur that have disjunctive sets of labels (no overlap at all) on outgoing transitions. The traditional learning bias often combines such states into one since it does not influence the acceptance of positive strings and inconsistencies due to negative strings (small deviations of positive behavior) are rare. The result, however, is a very different (much larger) language.
3. Even with a good bias, *it is unlikely that the best scoring DFA is identical to the target machine due to the data sparseness.* Ensuring this will typically require a large input set of diverse examples.

Any DFA identification technique has to face these problems when trying to synthesize software models. One of the goals of the StaMinA DFA learning competition was to find DFA learning techniques that perform well in this challenging setting. The EDSM algorithm performs very poorly on the StaMinA problem instances, achieving an accuracy of only 52% on the most difficult ones. Using our SAT-based method, we improved the accuracy on these problems to 95%. Significantly improving the state-of-the-art in software model DFA identification, and (partially) solving the problems faced when synthesizing software models.

Our algorithm is a combination of the greedy and exact DFA identification techniques described in the previous sections. It can be succinctly described as follows. We first apply greedy EDSM steps. Every such step (iteration) reduces the size of the remaining DFA identification problem. When the remaining problem is sufficiently small, we apply the SAT-based exact method. This solves the first problem, but unfortunately it also makes the result dependent on the EDSM heuristic. Since this heuristic is not well suited to learning software models (shown by the low accuracy scores), we develop a new heuristic and new merge restictions for EDSM that favor a large overlap in the labels of outgoing transitions of merged states, partially solving the second problem. Furthermore, since any greedy heuristic method can perform merges that lead to suboptimal solutions, we randomize the new evidence value and run it several times before returning the best found solution. Using such a random greedy procedure, we can actually find many possible

candidate solutions. Especially on the sparse problems, we cannot be sure which one of these DFAs is best. We therefore apply an ensemble method (Dietterich, 2000) in order to generalize these DFAs into an "average" DFA language. This partially solves the third problem.

Although all these techniques radically modify our algorithm, the SAT solver still serves as the core problem solving engine. Below we describe the different techniques in detail.

5.1 Greedy before search

The first problem is the most pressing one, because otherwise the SAT solver crashes or starts to swap memory[1]. We tackle it using the fact that our encoding is also valid for a *partially identified DFA* instead of an APTA. Since a partially identified DFA potentially combines (merges) many states of the APTA into one, this can significantly reduce the size of the encoding. The price to pay is of course that the solution provided by the SAT solver will no longer be exact. This now depends on whether the partial DFA is identified correctly. For identifying this partial DFA, we use the state-of-the-art EDSM algorithm in the red-blue framework. Our SAT-based DFA identification algorithm then works as follows:

1. Identify a partial DFA A' by applying EDSM for several iterations.
2. Apply a SAT solver to the remaining problem of finding the minimal consistent DFA A that contains A'.

Intuitively, using the red-blue framework for identifying A' is a natural choice since the first couple of merges are then based on evidence from many examples with respect to the number of available options (merges). Consequently, during these first merges, the heuristic value is well-estimated and therefore has a high probability of being correct, i.e., it is likely to lead to the optimal solution. With every merge that EDSM performs, more options and less examples become available, reducing this probability. An interesting open question is whether it is possible to compute this probability and use it as a guidance for when to switch to the exact solver. This could lead to DFA identification algorithms with high performance guarantees.

The decision value we used to determine when to switch during the StaMinA competition is based on practical considerations. In particular, for reasons that will become apparent in the following sections, we needed to be able to construct many small (not necessarily minimal) consistent DFAs in very little time. We therefore based the decision value on the size of the encoding of the remaining problem. This size is for a large part determined by the number of remaining non-red states reached by positive strings (the not pure negative parts of the original APTA that are not yet merged with red states). We switch once this value becomes less than 1000. We disregard the pure negative states since many of these will be merged into a rejecting sink (most of the negative strings from the StaMinA competition problems end up in such a sink). The remaining problem after this switch is typically solved in about a minute by the SAT solver.

[1] The machine we used for the competition contained 20Gb of RAM.

A bonus of first applying the EDSM algorithm in the red-blue framework is that we automatically obtain a clique of conflicting states: no red state can be merged with another red state. On the StaMinA problems, this clique is never smaller than the one found by the greedy max-clique algorithm. Therefore, we use these red states instead of the clique to construct the symmetry breaking predicates in our SAT encoding.

5.2 A new heuristic

The greedy before search method completely solves the first problem. Unfortunately, this method also makes the result very dependent on the heuristic used by EDSM. Due to the second problem, this heuristic is not well-suited for software model synthesis. Most importantly, it does not include any measure for the overlap in *positive fanout* of states. The positive fanout of a state is the set of symbols of outgoing transitions that can be activated by an accepting computation. In software models almost every state has a small positive fanout. We denote by $pf(q)$ the positive fanout of state q, i.e., $pf(q) = \{t \in T \mid \exists s \in S_+ \text{ with computation } q_0 t_1 \dots q t \dots q_n\}$. Moreover, since software models typically use a large alphabet of possible symbols, only a few of these states have exactly the same positive fanout. Therefore, if two states q and q' in the APTA have the same (or similar) positive fanout, this is an important indication that q and q' are actually the same state in the original software model that generated the data.

EDSM should thus use an evidence measure that favors merging states with a high degree of overlap in positive fanout. Hence a merge of q and q' is favorable if $|pf(q) \cap pf(q')| - |pf(q) \cup pf(q')|$ is small. We provide such a measure by counting the amount of positive merges. Let $|A|^p$ denote the number of states in A that are reached by the positive examples from the input sample, i.e., $|A|^p = |\{q \in Q \mid \exists s \in S_+ \text{ with computation } q_0 t_1 \dots q \dots q_n\}|$. For instance, in the APTA of Figure 3, from all states except q_5 an accepting state can be reached. The new evidence score is computed as follows:

$$\mathsf{new\_evidence}(A, q, q') := \begin{cases} |A|^p - |\mathsf{merge}(A, q, q')|^p & \text{if } \mathsf{merge}(A, q, q') \neq \text{FALSE} \\ -1 & \text{otherwise} \end{cases}$$

In other words, the difference in states reached by positive examples before and after the merge. This measure is computed by counting the number of merges between states that are both reached by a positive example. When there is a lot of overlap in positive fanout between the merged states, our new evidence measure obtains a high value because many children of these states are reached by positive examples, and will be merged during the determinization process. Which states are reached by positive examples can be computed efficiently by setting a Boolean flag to true in each state reached by a positive string. This flag can be set during APTA construction and maintained using the or-operation when merging states.

In contrast to the traditional EDSM measure, which only uses counts based on the rejecting and accepting states, our new evidence measure uses counts based on the entire computation of positive examples. It thus uses more information from the positive examples, and it completely disregards all information in the negative examples. This makes sense in software model synthesis because the behavior of

a software system is captured in the positive examples, representing correct executions of the system. The negative examples are simply random deviations of this behavior that represent faulty or incomplete executions. Due to this randomness, the computations of negative examples do not hold a lot of information. The only information from the negative examples used in our algorithm is that (like in EDSM) merges are required to be consistent, i.e., positive states and negative states cannot be merged.

5.3 Random greedy

The heuristic value described above gives preference to merges between states with a lot of overlap in positive fanout. Although this makes sense in software model synthesis, it is still a heuristic and it will not always select the most optimal merge, leading to the best DFA software model. It thus makes sense to sometimes try different merges, leading to different greedy paths, and resulting in different DFA software models. The best DFA model resulting from these different paths can then be returned as a solution. Unfortunately, it is not clear when to try different merges. Intuitively, it makes sense to try other merges that also obtain a high heuristic value, but how big the difference from the optimal value should be is unknown. A simple but effective technique called *random greedy* can be employed in such cases. This transforms the heuristic value in the following way:

$$\mathsf{random\_greedy} := \mathsf{random}() \cdot \mathsf{new\_evidence}$$

where $\mathsf{random}()$ is a function that return a value between 0.0 and 1.0 drawn uniformly at random. Using this transformation, the greedy procedure sometimes tries merges with heuristic values that are smaller than optimal. Additionally, if this value is much smaller than optimal, the probability that the merge is performed is very small. This makes sense intuitively, and works very well in practice. A potential problem occurs when many merges with small heuristic values are possible. In this case, the probability of performing a merge with a high heuristic value is small. We solve this problem by restricting the set of possible merges, and by repeating the greedy procedure many times. In the remainder of this section, we first describe these restrictions and then how we combine the result of many greedy runs.

5.4 Additional merge restrictions

In EDSM, the only restriction on a possible merge is that it is not allowed to be inconsistent (merging an accepting state with a rejecting state). Other than that, the decision on which merge to perform is determined using the evidence value. In software models, not only the acceptance condition of states, but also their positive fanout is important. In fact, in our experience with the StaMinA competition instances, the overlap in positive fanout turned out to be so important that we also use it in a consistency check by adding to $\mathsf{merge}\ (A, q, q')$:

if $q \in R$ and $pf(q') \not\subseteq pf(q)$ **or** $q' \in R$ and $pf(q) \not\subseteq pf(q')$ **then return** A

In other words, a merge is not allowed to add a new label to the positive fanout of a red state. The intuition behind this check is that, in the red-blue framework, the red states are assumed to be correctly identified. Therefore, we disallow adding new possible executions to the software system represented by these red states. Adding this consistency check turned out to be extremely important in solving the StaMinA problem instances.

Another new merge restriction involves an important property of software models, namely, that negative strings contain little information. This causes some problems when the greedy procedure starts to consider merging states with very small heuristic values. Although states reached only by negative strings always get a heuristic value of 0, it is possible that no good merge can be performed. In this case, it is always better to color a state red then to perform a bad merge. Since introducing new red states increases the number of possible merges, it is possible that good merges can again be found after coloring a state red. Coloring one of the pure negative states red, however, does not have this positive effect on later merges. Instead, it only increases the number of possible bad merges. Therefore, we do not even color pure negative states blue since this excludes them from being merged except by determinization. At the end of the greedy procedure, however, many transitions will now go from red states to uncolored pure negative states. Before calling the SAT solver, we merge these pure negative children of red states into a *rejecting sink* (a rejecting state with all outgoing transitions pointing to itself). Although it is possible that the exact solver now requires a few additional states to solve the problem (if some of these rejecting sink merges are suboptimal), this is worth it because merging these states into a single sink significantly reduces the size of the encoding.

5.5 An ensemble of automata

The previous three techniques and modifications all involved the greedy procedure and how to change its learning bias into the direction of software models, solving problem 2 from the beginning of this section. It could also be a good idea to modify the learning bias of the subsequent exact procedure. This would require a radically different encoding however, and since our current encoding (that minimizes the number of required additional states) works very well, we did not change this part of our algorithm. Instead, we focus on the remaining problem number 3, which is the sparseness of the available data.

Due to the sparseness of the data, it is unlikely that our algorithm finds the target DFA that was used to generate the data. During the StaMinA competition, we often encountered problem instances for which our algorithm found hundreds of different good solutions but none of them was good enough to solve the problem (achieving the required 99% accuracy). Surprisingly, some instances were suddenly solved when we combined all of these solutions:

$$\text{given a test string } s\text{, output } \begin{cases} \text{`1'} & \text{if at least 50\% of the DFAs accept } s \\ \text{`0'} & \text{otherwise} \end{cases}$$

Such an *ensemble* (Dietterich, 2000) of good automata seems to generalize really well over the different DFA languages, solving several StaMinA instances. Later,

we modified the condition to at least 10% since this increased the performance, solving several additional problem instances. Essentially, this means that the found DFAs have a tendency to reject strings. Investigating this tendency would be a very interesting direction for future work on software model synthesis.

6 The final algorithm

The final algorithm we used in the StaMinA competition is a combination of greedy techniques and our exact SAT translation. We believe this *combination of greedy and exact techniques* to be an important step forward in DFA identification. The key insight is that by iteratively introducing new states of the identified DFA, a greedy learning method continuously divides the data over more and more states. EDSM in the red-blue framework does so by coloring more and more states red. Consequently, with every iteration of the algorithm less data becomes available. Since the EDSM heuristic is based on statistics, this leads to a drop in performance. At some point, the heuristic performs so poorly that it makes sense to switch to an exact strategy.

Other important key points of our final algorithm are the *new heuristic* and *new consistency checks*. These effectively change the bias of a state-merging method in the direction of software models. The resulting increase in performance clearly shows the need to change this bias. We believe that such a need also exists in other applications of grammatical inference such as bio-informatics and computational linguistics, since, like software models, the models in these domains typically require large alphabets. It would be interesting to see what kind of heuristics performs well in these domains. We would like to point out, however, that even our new heuristic will perform poorly at some point. Therefore the combination with an exact technique such as a SAT solver is crucial for obtaining good performance.

Finally, the *ensemble of automata* is an interesting technique that works really well on sparse problem instances. Here, the key is to only combine *good solutions*. Therefore, instead of using every solution found by the random greedy algorithm, we only use those of small size. Furthermore, by decreasing this size bound (discussed below), we actively try to increase the quality of these solutions over time. Given enough good solutions (DFAs), we generalize them using a voting scheme that accepts a string if a substantial minority of the given DFAs accepts it (and rejects it otherwise). The first solutions can be discarded during the voting scheme to slightly improve the resulting generalization.

Our final algorithm (Algorithm 4) works in three steps:

1. First merge using EDSM, continue using SAT.
2. Fine-tune the target size.
3. Generalize over the found solutions.

We now explain each of these three steps, their parameters, and the settings used during the StaMinA competition.

EDSM vs SAT Ideally, one wants to solve a DFA identification problem exact (optimal value). In prior work (Heule and Verwer, 2010), we showed that SAT solving can beat EDSM implementations on hard instances. However, for the StaMinA

Algorithm 4 dfasat

```
Require: an input sample S, a test sample S_t, merge bound m, number of solutions n,
  accepting vote percentage avp between 0 and 1
Ensure: L is a labeling for S_t aimed to give high accuracy for software models
  let t := ∞                                          // the size bound t is initialized to infinity
  let D := ∅                                                    // D is an empty set of DFAs
  A = apta(S)                                                   // construct the APTA A
  while |D| < n do                                      // while D contains less than n DFAs
    let A' := copy(A)                                          // create a copy A' of the APTA
    while |A'|^p > m do       // while the positive strings reach more than m states in A'
      use random_greedy to select q and q' in A'
      call merge(A', q, q')                          // merge states in A' using random greedy
    end while

    if |R| > t (R being the red states in A') then          // if A' has more than t red states
      continue the next while loop iteration          // try to find a better partial solution
    end if
    set t := |R|                          // else update t to the amount of resulting red states

    let i := 0                                    // initialize the number of additional states to 0
    while true do          // while no solution has been found for the remaining problem
      translate A' using |R| + i colors     // try to find an exact solution with i extra states
      solve the formula using a SAT-solver
      if the solver returns a DFA solution A'' then
        add A'' to D and break              // if the SAT solver finds a solution add it to D
      else if the solver used the 300 seconds timeout then
        break                       // try another partial solution if the problem is too hard
      else
        set i := i + 1                                    // else try to find a larger solution
      end if
    end while
  end while

  let L be an empty labeling                                    // initialize the test labeling
  for all s ∈ S_t do                                                      // forall test strings
    if |{A ∈ D | s ∈ L(A)}| ≥ n · avp then
      append '1' to L   // label s positive if at least avp percent of the DFAs in D accept s
    else
      append '0' to L                                          // label s as negative otherwise
    end if
  end for
  return L
```

problems as well as many others, exact solving is not possible. Hence we propose to combine both alternatives.

Two main reasons make exact solving difficult. First, the merge restrictions, discussed in Section 5.4, are important for making the first merges effective. Our SAT translation does not cover concepts such as "red states" and "positive fanout". Adding these concepts, to encode these restrictions, will probably increase the size of the encoding significantly. Therefore, it seems best to perform the first merges using EDSM.

Second, the size of the APTA is simply too large for SAT solvers to deal with. Current state-of-the-art SAT solvers can solve structured problems, such as our DFA encoding, up to a few million clauses in reasonable time. Without applying some merge steps in advance, the SAT translation of the StaMinA problems con-

sists of hundreds of millions of clauses – definitely too big to solve. Hence, at least several merge steps are required to bring the size down.

The question arises, when to stop merging and start SAT solving? Within the algorithm, we use the parameter m for this purpose. It stops the merging sequence as soon as the size of the number of states in A that are reached by the positive examples from the input sample is smaller than m ($|A'|^p \leq m$). For the StaMinA competition we used the following reasoning to fix m. The main idea is to switch to SAT solving as soon as possible. So, when the APTA becomes small enough such that the translation consists of at most a few million clauses, then make the translation and apply SAT. We fixed $m := 1000$ because that value satisfied this objective.

After StaMinA finished, detailed results became available. We therefore also experimented with smaller values of m to see whether the performance of the algorithm decreased. It appeared that this was hardly the case. It is possible to decrease m to approximately 300 without any measurable performance loss. Apparently, random greedy EDSM can still make (almost) the optimal merges up to that point for some of the merge sequences. The advantage of setting m to a lower value is that the algorithm will generate solutions faster. This is especially useful in case one wants to generate many solutions in order to generalize them (discussed below).

Tuning the target size. In a partially identified DFA A, the red states R form the part of A that is assumed to be correctly identified. Therefore, the smallest DFA A' that can be found by the SAT solver when starting from A has at least r states. Now, consider two partially identified DFAs that both have (almost) the same size, but they differ in the number of red states. It is expected that one can construct a smaller DFA solution when starting from the one with fewer red states. Therefore we use the number of red states as a measurement on the effectiveness of a random greedy sequence.

This measurement is used as follows in the dfasat algorithm. A parameter t is used that refers to the target size of the DFA. If after merging, the number of red states $|R|$ is larger than t, then that merge sequence is considered as ineffective. Initially, $t := \infty$. So, no matter how bad the merge sequence, the remaining part is solved exactly by the SAT solver. The target size t is decreased over time: If the SAT procedure is called ($|R| \leq t$), then t is reduced to $|R|$ ($t := |R|$). As a result of the decrement of t over time, the size of the generated DFA solution is expected to decrease as well. There is no strong relation between the number of red states of a partially identified DFA A and the smallest DFA A' found starting from A'. For the StaMinA problems we observed that in most cases the size of A' is between $|R|$ and $|R| + 6$.

In practice, t converges fast. For the StaMinA benchmarks, t was close to the converging value after a few decrements. Yet different values were observed for different problems. The value of t after a few decrements ranged between 30 and 130. All problems of the StaMinA competition were constructed using a DFA of approximately size 50. So the large range of values can hardly be explained based on the construction method. In general (but with exceptions), the sparser the data, the larger t, while the larger the alphabet, the smaller t.

Generalizing over the found solutions. The algorithm (Algorithm 4) generates many solutions. For the StaMinA problems with 100% sparsity, one of the first solutions had an accuracy of 99% thereby solving the problem. However, for almost all the problems with less than 100% sparsity, a single solution (even when generating dozens of them) did not have an accuracy of 99%. In order to solve several of these problems we developed a technique to combine good solutions.

Once the algorithm has found quite some (say $n := 100$) good complete solutions, we let each of these stored DFAs determine whether it accepts or rejects the strings in the test set. In our experience, using 100 good solutions is sufficient to get good results and more solutions hardly improved the result further. Every string from the test set that is accepted by a "substantial minority" of the DFAs gets assigned a positive label '1', all other strings get a negative label '0'. We refer to the accepting vote percentage (*avp*) as this fraction of the "substantial minority". For the competition we used 10% of the DFAs as *avp*, because using it solved most instances.

After the competition, when detailed results became available, it appeared that a voting scheme depending on 10% of the DFAs is hardly the best one in general. Table 2 shows the results for different vote schemes based on the *avp*. We selected ten instances for which i) dfasat performs reasonably well and ii) the voting scheme has an observable influence. The size of the alphabet has a clear impact on the optimal voting scheme. For an alphabet of size two, a voting scheme that requires 50% of the DFAs to accept a string was optimal. With increasing sizes of the alphabet, the voting scheme should accept a string if fewer DFAs accept that string. The most extreme example that we encountered was problem #91. For this problem, only a voting scheme requiring 1% of the DFAs to accepted it resulted in a BCR (accuracy, see next section) of 98.

Table 2 Balanced Classification Rate (BCR) for selected problems using different voting schemes based on the *avp*. The bold values show the highest rate per problem.

problem	$\|L\|$	10%	20%	30%	40%	50%	60%	70%	80%	90%
#8	2	89	92	94	**95**	**95**	**95**	94	93	91
#14	2	95	97	**98**	**98**	**98**	**98**	**98**	97	93
#26	5	94	**97**	**97**	**97**	96	95	94	92	88
#31	5	84	87	**89**	**89**	84	78	67	57	45
#49	10	96	**97**	**97**	96	94	92	88	84	78
#55	10	97	97	**98**	97	97	95	90	84	77
#69	20	**98**	**98**	**98**	**98**	**98**	97	97	96	96
#73	20	**99**	**99**	**99**	98	98	97	96	92	86
#86	50	**99**	98	97	97	97	97	96	96	95
#91	50	**97**	96	95	94	94	93	92	89	86

7 Results

We implemented the dfasat algorithm while focusing on fast performance on the StaMinA competition benchmarks[2]. Our implementation strongly follows the pseudocode as shown in Algorithm 4. Some minor tweaks discussed below were applied

[2] available from `http://stamina.chefbe.net/download`

to speed up the algorithm for the benchmark set. The picosat solver (Biere, 2008) was selected to tackle the resulting CNF formulas. This solver appeared to be the fastest around for this application.

7.1 Running dfasat on StaMinA

The CNF formulas that were generated by our translator consisted on average of a few million clauses for each of the StaMinA competition problems. These formulas include all redundant clauses (Section 4.2) and symmetry breaking predicates (Section 4.4). Since the latter are mostly unit clauses, the translator simplified the formula during the translation (otherwise the output would have been significantly larger). SAT solvers would perform a similar simplification, but the translator is much faster than writing the huge formula to disk and let the SAT solver deal with it.

Although formulas with a few million clauses may appear hard to solve, in practice most were relatively easy. The satisfiable formulas (with solutions) were solved between a few seconds and five minutes. The unsatisfiable formulas (without solutions, so the target DFA is too small) required clearly more time; between a minute and an hour. Because unsatisfiable formulas do not result in a DFA, we killed the SAT solver after 5 minutes to speed up the whole process. For the 25 competition instances with 100% sparsity, the dfasat algorithm could quickly find a DFA which labeled the test set with an accuracy of at least 99% according to the online competition oracle. For most of these instances, just a few DFAs were computed of which one passed the accuracy test. For the harder ones a dozen DFAs were required to pass the test using the 10% voting scheme. Our implementation could solve all the 100% sparsity instances in less than 15 minutes per problem.

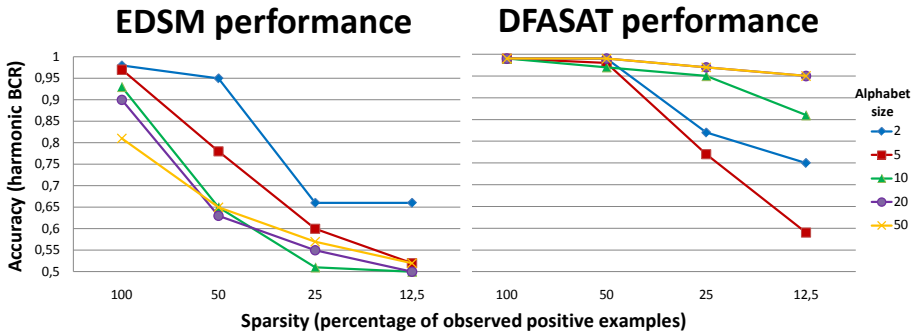
The problems with 50% sparsity required more computational cost. For each of those problems we ran the algorithm for one hour per instance and submitted the labeling of the test set using the 10% voting scheme. Using this approach, we were able to solve about half of the instances. For the unsolved instances, we ran the algorithm for an additional three hours per instance which helped to solve a few more instances. For problem #8 (the smallest unsolved benchmark) we even ran the algorithm for 24 hours. Yet it remained unsolved. For all 50 instances with very low sparsity (25% or 12.5%) we ran the algorithm for an hour and submitted the result using the 10% voting scheme. Based on the experience with 50% sparsity, we had no expectation to solve any of these instances.

7.2 Final results

The results of our algorithm on the StaMinA problems are shown in Figure 7. The accuracy score is computed according to the (BCR) measure, which is the harmonic mean of the true positive and true negative ratios, see (Walkinshaw et al, 2010):

$$\mathsf{BCR} = \frac{2 \cdot \mathsf{sensitivity} \cdot \mathsf{specificity}}{\mathsf{sensitivity} + \mathsf{specificity}}$$

where sensitivity is $\frac{TP}{TP+FN}$, specificity is $\frac{TN}{TN+FP}$, and TP, TN, FP, FN are the true and false positives and negatives.



	EDSM				DFASAT			
	100	50	25	12,5	100	50	25	12,5
2	0,98	0,95	0,66	0,66	0,99	0,99	0,82	0,75
5	0,97	0,78	0,6	0,52	0,99	0,98	0,77	0,59
10	0,93	0,65	0,51	0,5	0,99	0,97	0,95	0,86
20	0,9	0,63	0,55	0,5	0,99	0,99	0,97	0,95
50	0,81	0,65	0,57	0,52	0,99	0,99	0,97	0,95

Fig. 7 The resulting average Balanced Classification Rate values of EDSM and dfasat on the StaMinA problem instances.

In addition to winning the StaMinA competition, the figure shows we also improved the accuracy of the state-of-the-art grammatical inference method for software synthesis from an average performance of 52% (using EDSM) to 95% on the hardest problems. Our technique thus shows big potential for solving real-world software synthesis problems.

Figure 7 also shows that our technique does not perform well on sparse problem instances with small alphabets. This is partially due to the new heuristic. The heuristic considers the overlap in positive fanout to be the most important criterion when deciding which merge to perform. Since this is less important when the alphabet is small, using another heuristic such as the original EDSM evidence value potentially improves our performance on these instances. Furthermore, as shown in the previous section, the 10% voting scheme later turned out to be unsuited for problems with a small alphabet.

Due to the construction method of the StaMinA instances[3], the training sets contain a lot of duplicate examples (especially positive ones). As a result, the smaller the alphabet, the larger the fraction of duplicates. Hence, competition instances with larger alphabets have more unique training examples. Since the examples used for testing do not overlap with the training examples, we expect that this contributed to the performance of dfasat on larger alphabets.

8 Conclusions

We presented dfasat, the winning solver of the StaMinA DFA learning competition. Our contributions are the following:

– We provide an *efficient translation* from DFA identification into satisfiability.

[3] see `http://stamina.chefbe.net/machines`

– We show how to use a *greedy state-merging algorithm* before applying the *exact SAT solver*.
– We give a *new heuristic* and *new consistency checks* that are dedicated to software model synthesis problems.
– We demonstrate the effectiveness of *random greedy* and *ensemble methods* in DFA identification problems.

With our technique, the performance of the state-of-the-art on the hardest StaMinA problems (a size 50 alphabet and observing only 12.5% of the input examples) is increased from 52% to 95%. This shows the big potential of our technique for solving real-world software synthesis problems. Furthermore, since many other application areas such as bio-informatics and computational linguistics typically also require a large alphabet, it would be interesting to test the performance of our method in these domains as well.

Our technique is a unique *combination of exact and greedy techniques* that we believe has a lot of merit in many machine learning problems. The key insight of our method is that greedy methods in machine learning perform well when a lot of data is available since they are typically based on statistics. Furthermore, with every step, a greedy method typically divides the data over newly identified elements of a model. An example is decision tree learning in which the greedy method divides the data over new leaf nodes in every step. In every subsequent iteration of such a greedy method less and less data becomes available. At some point, the statistics in the heuristic will be poorly estimated and therefore it makes sense to switch to an exact strategy.

Using a SAT solver for the exact part of our algorithm seems a very good choice since it provides many advanced solving techniques. In addition, since less data becomes available with every greedy iteration, we can perform greedy steps until the size of the SAT encoding is relatively small. The SAT solver then requires little time to decide whether the remaining problem is 'satisfiable'. In this way, it is possible to *generate many good solutions in just a few minutes*. An ensemble of these solutions nearly solved the most difficult StaMinA problems. We are very interested to see how our technique performs in different problem domains. The code is available online at `http://www.st.ewi.tudelft.nl/sat/dfasat.php`.

References

van der Aalst WMP (2011) Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer

Aarts F, Vaandrager F (2010) Learning I/O automata. In: CONCUR, Springer, Lecture Notes in Computer Science, vol 6269, pp 71–85

Abela J, Coste F, Spina S (2004) Mutually compatible and incompatible merges for the search of the smallest consistent DFA. In: ICGI, Springer, LNCS, vol 3264, pp 28–39

Ammons G, Bodik R, Larus JR (2002) Mining specifications. In: Proceedings of the Symposium on Principles of Programming Languages, ACM, pp 4–16

Antunes J, Neves N, Verissimo P (2011) Reverse engineering of protocols from network traces. Reverse Engineering, Working Conference on 0:169–178

Bertolino A, Inverardi P, Pelliccione P, Tivoli M (2009) Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of the joint meet-

ing of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, pp 141–150
Biere A (2008) PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 4(2-4):75–97, URL http://jsat.ewi.tudelft.nl/content/volume4/JSAT4_5_Biere.pdf
Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: TACAS '99, Springer, London, UK, pp 193–207
Biere A, Heule MJH, van Maaren H, Walsh T (eds) (2009) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, Handbook of Satisfiability, vol 185. IOS Press
Biermann AW, Feldman JA (1972) On the synthesis of finite-state machines from samples of their behavior. IEEE Trans Comput 21(6):592–597, DOI http://dx.doi.org/10.1109/TC.1972.5009015
Broy M, Jonsson B, Katoen JP, Leucker M, Pretschner A (eds) (2005) Model-Based Testing of Reactive Systems, Lecture Notes in Computer Science, vol 3472, Springer
Bugalho M, Oliveira AL (2005) Inference of regular languages using state merging algorithms with search. Pattern Recognition 38:1457–1467
Castro J, Gavaldà R (2008) Towards feasible PAC-learning of probabilistic deterministic finite automata. In: ICGI, pp 163–174
Clark A, Thollard F (2004) PAC-learnability of probabilistic deterministic finite state automata. Journal of Machine Learning Research pp 473–497
Clarke E (1997) Model checking. In: Foundations of Software Technology and Theoretical Computer Science, LNCS, vol 1346, Springer, pp 54–56
Cook JE, Wolf AL (1998) Discovering models of software processes from event-based data. ACM Trans Softw Eng Methodol 7:215–249
Coste F, Nicolas J (1997) Regular inference as a graph coloring problem. In: Workshop on Grammatical Inference, Automata Induction, and Language Acquisition (ICML'97)
Cui W, Kannan J, Wang HJ (2007) Discoverer: automatic protocol reverse engineering from network traces. In: Proceedings of 16th USENIX Security Symposium, p nr. 14
Dallmeier V, Lindig C, Wasylkowski A, Zeller A (2006) Mining object behavior with ADABU. In: Proceedings of the 2006 international workshop on Dynamic systems analysis, ACM, WODA '06, pp 17–24
Denis F, Lemay A, Terlutte A (2000) Learning regular languages using non deterministic finite automata. In: ICGI, pp 39–50
Dietterich T (2000) Ensemble methods in machine learning. In: Multiple Classifier Systems, LNCS, vol 1857, Springer, pp 1–15
Dupont P, Lambeau B, Damas C, van Lamsweerde A (2008) The QSM algorithm and its application to software behavior model induction. Applied Artificial Intelligence 22(1&2):77–115
Endrullis J, Waldmann J, Zantema H (2008) Matrix interpretations for proving termination of term rewriting. J Autom Reason 40(2-3):195–220, DOI http://dx.doi.org/10.1007/s10817-007-9087-9
Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA
Gold EM (1978) Complexity of automaton identification from given data. Information and Control 37(3):302–320

Grinchtein O, Jonsson B, Petterson P (2006a) Inference of event-recording automata using timed decision trees. In: CONCUR, Springer, LNCS, vol 4137, pp 435–449

Grinchtein O, Leucker M, Piterman N (2006b) Inferring network invariants automatically. In: Automated Reasoning, Springer, LNCS, vol 4130, pp 483–497

Heule MJH, Verwer S (2010) Exact DFA identification using SAT solvers. In: ICGI, Springer, LNCS, vol 6339, pp 66–79

de la Higuera C (2010) Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, New York, NY, USA

Higuera Cdl, Janodet JC (2004) Inference of omega-languages from prefixes. Theoretical Computer Science 313(2):295–312

Jain S, Osherson D, Royer JS, Sharma A (1999) Systems that learn. MIT Press

Jarvisalo M, Biere A, Heule MJH (2010) Blocked clause elimination. In: TACAS 2010, Springer, LNCS, vol 6015, pp 129–144

Juillé H, Pollack JB (1998) A sampling-based heuristic for tree search applied to grammar induction. In: Innovative applications of artificial intelligence, American Association for Artificial Intelligence, Menlo Park, CA, USA, AAAI '98, pp 776–783, URL `http://dl.acm.org/citation.cfm?id=295240.295804`

Kearns MJ, Vazirani UV (1994) An introduction to computational learning theory. MIT Press

Kullmann O (1999) On a generalization of extended resolution. Discrete Applied Mathematics 96-97(1):149–176, DOI http://dx.doi.org/10.1016/S0166-218X(99)00037-2

Lang KJ (1999) Faster algorithms for finding minimal consistent DFAs. Tech. rep., NEC Research Institute

Lang KJ, Pearlmutter BA, Price RA (1998) Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: ICGI, Springer, LNCS, vol 1433

Mariani L, Pastore F, Pezze M (2011) Dynamic analysis for diagnosing integration faults. IEEE Transactions on Software Engineering 37:486–508

Marques-Silva JP, Glass T (1999) Combinational equivalence checking using satisfiability and recursive learning. In: Proceedings of DATE '99, ACM, New York, NY, USA, p 33, DOI http://doi.acm.org/10.1145/307418.307477

Oliveira AL, Marques-Silva JP (1998) Efficient search techniques for the inference of minimum sized finite state machines. String Processing and Information Retrieval pp 81–89

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

Pitt L, Warmuth M (1989) The minimum consistent DFA problem cannot be approximated within and polynomial. In: Annual ACM Symposium on Theory of Computing, ACM, pp 421–432

Raffelt H, Steffen B, Berg T, Margaria T (2009) Learnlib: a framework for extrapolating behavioral models. International Journal on Software Tools for Technology Transfer (STTT) 11:393–407

Sakallah KA (2009) Symmetry and Satisfiability, chap 10, pp 289–338. Vol 185 of Biere et al (2009)

Sudkamp TA (2006) Languages and Machines: an introduction to the theory of computer science, 3rd edn. Addison-Wesley

Verwer S (2010) Efficient identification of timed automata: Theory and practice. PhD thesis, Delft University of Technology

Walkinshaw N, Bogdanov K, Holcombe M, Salahuddin S (2007) Reverse engineering state machines by interactive grammar inference. In: Proceedings of the 14th Working Conference on Reverse Engineering, IEEE, pp 209–218

Walkinshaw N, Bogdanov K, Damas C, Lambeau B, Dupont P (2010) A framework for the competitive evaluation of model inference techniques. In: Proceedings of the First International Workshop on Model Inference In Testing, ACM, New York, NY, USA, MIIT '10, pp 1–9

Walsh T (2000) SAT v CSP. In: CP '02: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, Springer, London, UK, pp 441–456

Yokomori T (1993) Learning non-deterministic finite automata from queries and counterexamples. In: Machine Intelligence, University Press, pp 196–189