Applied Artificial Intelligence, 22:77–115
Copyright © 2008 Taylor & Francis Group, LLC
ISSN: 0883-9514 print/1087-6545 online
DOI: 10.1080/08839510701853200

THE QSM ALGORITHM AND ITS APPLICATION TO SOFTWARE BEHAVIOR MODEL INDUCTION

Pierre Dupont □ *Department of Computing Science and Engineering (INGI), Université Catholique de Louvain, Belgium, and the UCL Machine Learning Group*

Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde □ *Department of Computing Science and Engineering (INGI), Université Catholique de Louvain, Belgium*

□ *This article presents a novel application of grammatical inference techniques to the synthesis of behavior models of software systems. This synthesis is used for the elicitation of software requirements. This problem is formulated as a deterministic finite-state automaton induction problem from positive and negative scenarios provided by an end user of the software-to-be. A query-driven state merging (QSM) algorithm is proposed. It extends the Regular Positive and Negative Inference (RPNI) and blue-fringe algorithms by allowing membership queries to be submitted to the end user. State merging operations can be further constrained by some prior domain knowledge formulated as fluents, goals, domain properties, and models of external software components. The incorporation of domain knowledge both reduces the number of queries and guarantees that the induced model is consistent with such knowledge. The proposed techniques are implemented in the ISIS tool and practical evaluations on standard requirements engineering test cases and synthetic data illustrate the interest of this approach.*

INTRODUCTION

It has been claimed that the hardest part in building a software system is deciding precisely what the system should do (Brooks 1987). This is the general objective of requirements engineering (RE). Formal models are increasingly recognized as an effective means for elaborating requirements and exploring software designs. For complex systems, model building however is far from an easy task. Automating parts of this process can be addressed by learning behavior models from scenarios of interactions between the software-to-be and its environment. Indeed, scenarios can be

Address correspondence to Pierre Dupont, Department of Computing Science and Engineering (INGI), Université Catholique de Louvain, Place Sainte Barbe, 2, B-1348 Louvain-la-Neuve, Belgium. E-mail: Pierre.Dupont@uclouvain.be

represented as strings over an alphabet of possible events and they can be generalized to form a language of acceptable behaviors. Whenever behaviors are modeled by finite-state machines, the problem of behavior model synthesis becomes equivalent to automaton induction from positive and negative strings. The learning examples are assumed here to be positive and negative scenarios provided by an end-user involved in the requirements elicitation process. Whenever available, additional domain knowledge can also help the induction process and force the resulting model to be consistent with this knowledge.

In the present work, we address the problem of the synthesis of a behavior model of a software system by extending state-of-the-art automaton induction techniques: the RPNI (Oncina and García 1992) and blue-fringe algorithms (Lang, Pearlmutter, and Price 1998). Both algorithms can indeed profit from the presence of an end user. The proposed techniques are incremental, since they deal with a growing collection of examples, and interactive, since the end user may be asked to classify as positive or negative additional scenarios, which are automatically generated. In learning terminology, the end user is assumed to be an oracle who can answer membership queries (Angluin 1981).

The first section describes why the proposed application of grammatical inference techniques seems particularly relevant. Our main contributions and the organization of the rest of this article are summarized in the second section.

System Behavior Modeling and Automaton Induction

We argue here that automaton induction is a particularly well-suited approach to model software system behaviors. This application domain also offers potentially new perspectives for defining relevant domain knowledge to constrain the learning process.

Behaviors are conveniently modeled by labeled transition systems (LTS). The class of LTS models is properly included in the deterministic finite-state automaton (DFA) class, as detailed in Section 2.2. Hence, LTS synthesis is a DFA induction problem, which is arguably the most studied problem in grammatical inference.

Hardness results of the minimal consistent DFA identification problem are known (Angluin 1978; Gold 1978). However, positive results exist for approximate identification of randomly generated DFAs in the average case (Lang 1992). Extensions of the above technique, such as the blue-fringe or Evidence-Driven State Merging (EDSM) algorithms, were shown to be particularly effective in the context of the Abbadingo DFA induction competition (Lang et al. 1998). This competition and its successor Gowachin illustrate that available techniques can learn DFAs of several hundred states. A behavior model hardly contains such a large number of states.

Hence, current DFA learning techniques seem directly applicable as tools for software analysts.

The above applicability issue must be taken with a grain of salt however. Firstly, the positive results observed in DFA learning benchmarks were generally obtained with very small alphabets, typically containing two symbols. Standard test cases in requirements engineering, such as those detailed in Section 6, may deal with alphabets an order of magnitude larger. More importantly, according to the practical setting considered in the DFA learning competitions mentioned above, successful learning of DFAs with hundred states requires training sets containing typically several thousand strings. This is far beyond the available data one could reasonably ask an end user to provide. Increasing the alphabet size and reducing the available learning data both significantly increases the complexity of the induction problem (Lang et al. 1998).

Fortunately, the end user may also be asked to answer membership queries apart from the initial scenarios she provides. The presence of an oracle does not only change the theoretical limits of learning (Angluin 1981) but also forms a fully realistic assumption in the applicative context considered here. In contrast, the model of a minimally adequate teacher introduced by Angluin (1987) also assumes that the oracle can answer equivalence queries. In such a case, a candidate model is submitted to the oracle who has to decide whether it perfectly defines the target behavior. If it is not the case, the oracle is asked to return a counterexample.[1] We consider that the practical availability of such a well-informed end user is unrealistic. Interestingly, in practice, membership queries alone may very well be sufficient to learn an adequate model. This point is further discussed in Sections 6 and 7.

The induction algorithms considered here are state-merging techniques. They start from a machine that accepts strictly the positive examples initially provided. States of this original machine are subsequently merged under the control of the available negative examples. State merging generalizes the accepted language (see Section 3). Negative examples can also be represented in the original machine as accepting states with a negative label. Consistency with the positive and negative examples is guaranteed if the merging process is constrained to prevent merging a negative accepting state with a positive accepting state. In other words, the positive and negative information define equivalence classes between states. Only states belonging to the same equivalent class are allowed to be merged. This strategy can readily be extended to incorporate domain knowledge by refining the equivalence state partition (Coste et al. 2004). The novelty in our approach is the specific kinds of domain knowledge considered here, which is often available as declarative properties in the RE context, and the way they are reformulated as equivalence classes (see Section 5). Our practical evaluations reported in Section 6.1 illustrate that these additional constraints can significantly reduce the number of interactions with the end user, while enforcing the proposed model to be consistent with such prior knowledge. This extension could also be useful for other application domains whenever such declarative properties can be stated.

Outline

The rest of this article is structured as follows. Section 2 gives a brief introduction to a model-driven approach to the elaboration of software requirements. It includes a description of scenarios in terms of message sequence charts and the particular class of automata considered in this work. It also introduces a running example of a simple train system which is used throughout this article to illustrate the proposed techniques.

Section 3 recalls a few important notions relative to DFA induction. In particular, the notion of characteristic sample for the RPNI induction algorithm is reviewed. This characterization is subsequently used to define which additional scenarios should be submitted to be classified by an end user during the learning process.

Section 4 describes our first contribution: the QSM algorithm (a Query-driven State Merging induction approach). It is an interactive DFA learning algorithm that extends the RPNI algorithm by including membership queries. The blue-fringe algorithm can be considered as an adaptation of this state merging approach by changing the order in which state pairs are considered. The extension of our interactive methodology within the blue-fringe strategy is also discussed.

Section 5 details our second contribution: how domain knowledge, specific to the RE application context, can be included in the learning process. This domain knowledge is by no means mandatory for the inference process. However, when it is used, it speeds up the induction process and also guarantees that the proposed model is consistent with this domain knowledge.

The proposed techniques are implemented in Java 5.0 to form the Interactive State Machine Induction from Scenarios (ISIS) tool. Experimental evaluations of this tool are presented in Section 6. They illustrate that the use of the blue-fringe strategy and domain knowledge allows the reduction of the number of interactions with the end user. The first evaluations are performed on standard RE test cases (Section 6.1). They correspond to relatively small target machines (up to 23 states). The number of actual scenarios initially provided is also limited—typically less than 10—but the end user is asked to classify additional scenarios generated automatically. These test cases can be considered small from a learning perspective but we argue that they are fully representative of the existing RE literature. Hence, Section 6.2 addresses the question of how the proposed approach scales with the size of the target machine on synthetic data.

We address in particular the accuracy of the induced automata while varying the amount of training data. Comparative results are reported for the RPNI, blue-fringe, and QSM algorithms. The number of queries asked to the end user and the total CPU time of the induction process are also reported.

Related works are presented in Section 7, including a discussion on the theoretical limits and practical issues of learning with queries. Our conclusions and perspectives are presented in Section 8.

MODEL-DRIVEN ELABORATION OF SOFTWARE REQUIREMENTS

Software requirements have been repeatedly recognized to be a real problem. In their early empirical study, Bell and Thayer observed that inadequate, inconsistent, incomplete, or ambiguous requirements are numerous and have a critical impact on the quality of the resulting software (1976). Boehm estimated that the late correction of requirements errors could cost up to 200 times as much as correction during such requirements engineering (1981). In his classic article on the essence and accidents of software engineering, Brooks stated that "the hardest single part of building a software system is deciding precisely what to build" (Brooks 1987). Recent studies have confirmed the requirements problem on a much larger scale. A survey of over 8000 projects undertaken by 350 U.S. companies revealed that one-third of the projects were never completed and one-half succeeded only partially, that is, with partial functionalities, major cost overruns, and significant delays (The Standish Group 1995). When asked about the causes of such failure, executive managers identified poor requirements as the major source of problems.

Improving the quality of requirements is thus crucial. Requirements engineering is concerned with the elicitation, negotiation, specification, analysis, and evolution of the objectives, functionalities, qualities, and constraints to be achieved by a software system within some organizational or technical environment.

Model-driven elaboration, validation, and documentation of requirements call for rich models of the system-to-be. Such models need to cover the intentional, structural, and behavioral aspects of the system (van Lamsweerde 2001)—by system we mean both the target software and its environment. The intentional aspect captures the purposes and rationales of the system (goals). The structural aspect represents the concepts of the application domain (class diagrams) and the behavioral dimension defines how agents react to events (scenarios and state machines).

Goals are prescriptive statements of intent whose satisfaction requires cooperation among the agents forming the system. Goal models are AND/OR graphs that capture how functional and non functional goals contribute positively or negatively to each other. Such models support various forms of early, declarative, and incremental reasoning, e.g., goal refinement and completeness checking, conflict management, hazard analysis, threat analysis, requirements document generation, etc. (van Lamsweerde 2004). On the down-side, goals are sometimes felt too abstract by end users of the system-to-be. They cover classes of intended behaviors but such behaviors are left implicit. Goals may also be hard to elicit in the first place. A typical goal for a train system is to require that the train doors shall remain closed while the train is moving. This property is not guaranteed to be satisfied by some physical law and thus must be enforced as a system goal.

Scenarios capture typical examples or counterexamples of system behavior through sequences of interactions among agents. They support an informal, narrative, and concrete style of description. Scenarios are therefore easily accessible to end users involved in the requirements engineering process. On the down-side, scenarios are inherently partial and cover few behaviors of specific instances. They leave intended system properties implicit. Scenarios must thus be generalized to elicitate additional system properties.

State machines capture classes of required agent behaviors in terms of states and events firing transitions. They provide visual abstractions of explicit behaviors for any agent instance in some corresponding class. State machines can be composed sequentially and in parallel, and are executable. They can be validated through animation and verified against declarative properties. State machines also provide a good basis for code generation. On the down-side, state machines are too operational in the early stages of requirements elaboration. Their manual elaboration may turn out to be quite hard.

In this study, we show how a scenario-driven model synthesis can be achieved through the use of automaton learning techniques. Goals are introduced further as a particular type of optional domain knowledge that can constrain the learning process. Whenever goals are formulated they help to speed up learning and guarantee that the induced model is consistent with this knowledge. Section 2.1 describes the formalism of message sequence charts to represent scenarios, which can also be seen as strings over an alphabet of events. Section 2.2 introduces labeled transition systems, which form the particular class of automata considered throughout this article.

Scenarios and Message Sequence Charts

A scenario is a temporal sequence of interactions among system components. A system is made of active components, called agents, which control system behaviors. Some agents form the environment, others form the software-to-be. An interaction in a scenario originates from some event synchronously controlled by a source agent instance and monitored by a target agent instance. Positive scenario describe typical examples of desired interactions, whereas negative scenarios described undesired ones. The set of possible events defines an alphabet Σ and a scenario is an element of Σ^*, that is, a finite-length string defined over Σ. In the sequel, $|x|$ denotes the length of a string x belonging to Σ^* and λ denotes the empty string.

A simple message sequence chart (MSC) formalism is used for representing end user scenarios. An MSC is composed of vertical lines representing timelines associated with agent instances, and horizontal arrows representing interactions among such agents. A timeline label specifies the type of corresponding agent instance. An arrow label specifies some event defining the corresponding interaction. Every arrow label uniquely determines the source and target agent instances that control and monitor the event in the interaction, respectively. We choose a simple input language in order to allow end users to submit their scenarios, leaving aside more sophisticated MSC features such as conditions, timers, coregions, etc.

A simple train system fragment will be used throughout the article as a running example. The system is composed of three agents: a train controller, a train actuator/sensor, and passengers. The train controller controls operations such as `start`, `stop`, `open doors`, and `close doors`. A safety goal requires train doors to remain closed while the train is moving. If the train is not moving and a passenger presses the alarm button, the controller must open the doors in an emergency. When the train is moving and the passenger presses the alarm button, the controller must stop the train first and then open the doors in an emergency. Figure 1 shows a MSC capturing this last scenario.

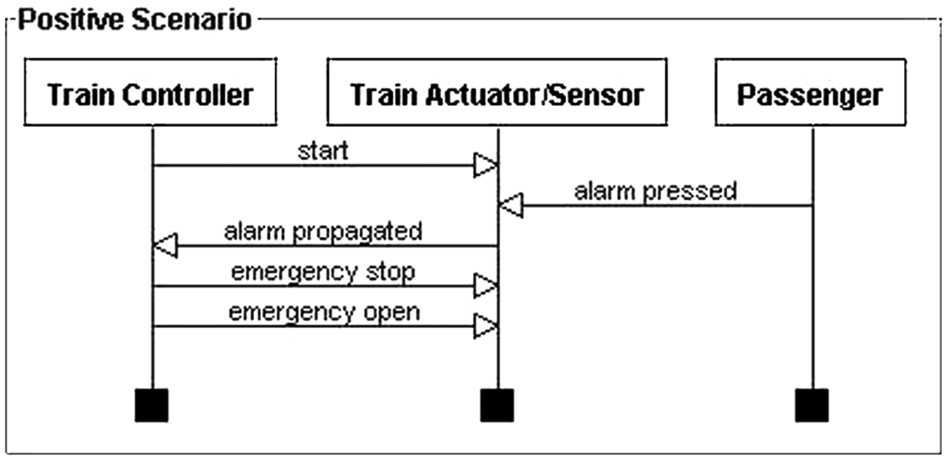

FIGURE 1 Positive scenario for a train system.

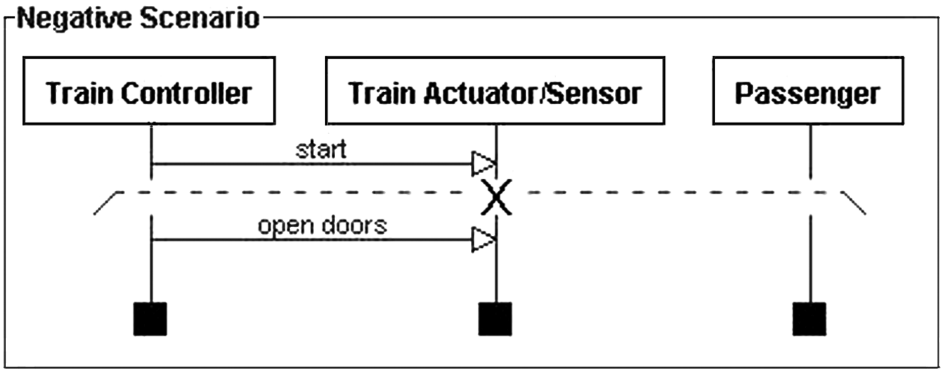

FIGURE 2 Negative scenario for a train system.

Scenarios are positive or negative. A positive scenario illustrates some desired system behavior. Any prefix of a positive scenario is also a positive scenario. A negative scenario captures a behavior that may not occur. It is represented by a pair (p, e), where p is a positive MSC called precondition, and e is a prohibited subsequent event. The meaning is that once the admissible MSC precondition has occurred, the prohibited event may not label the next interaction among the corresponding agents. A negative scenario is a string composed of the prefix $p \in \Sigma^*$, which forms a positive scenario, and a single event $e \in \Sigma$ such that the concatenated string pe should be labeled as negative.

Figure 2 shows a negative scenario. The MSC precondition is made of the interaction `start`; the prohibited event is `open doors`. Prohibited events in negative MSCs appear below a dashed line in our ISIS tool. The scenario in Figure 2 is used to express that the train controller may not open the doors after having started the train (without any intermediate interaction).

Starting from an initial set of positive and negative scenarios, additional scenarios are automatically generated during the learning procedure described in Section 4. Such scenarios are submitted to the end user to be classified as positive or negative. Typical examples produced by the ISIS tool are presented in Figure 3. The prefix (above the dashed line) p is already known to correspond to an acceptable behavior. The suffix must be accepted or rejected by the end user as a valid continuation. In practice, the end user is allowed to move down the dashed line to indicate that an extended prefix p' should be considered as positive as well and the first event e after p', if any, defines a negative scenario $p'e$.

Finite Automata and Labeled Transition Systems

A system can be behaviorally modeled as a set of concurrent state machines—one machine per agent. Each agent is characterized by a set

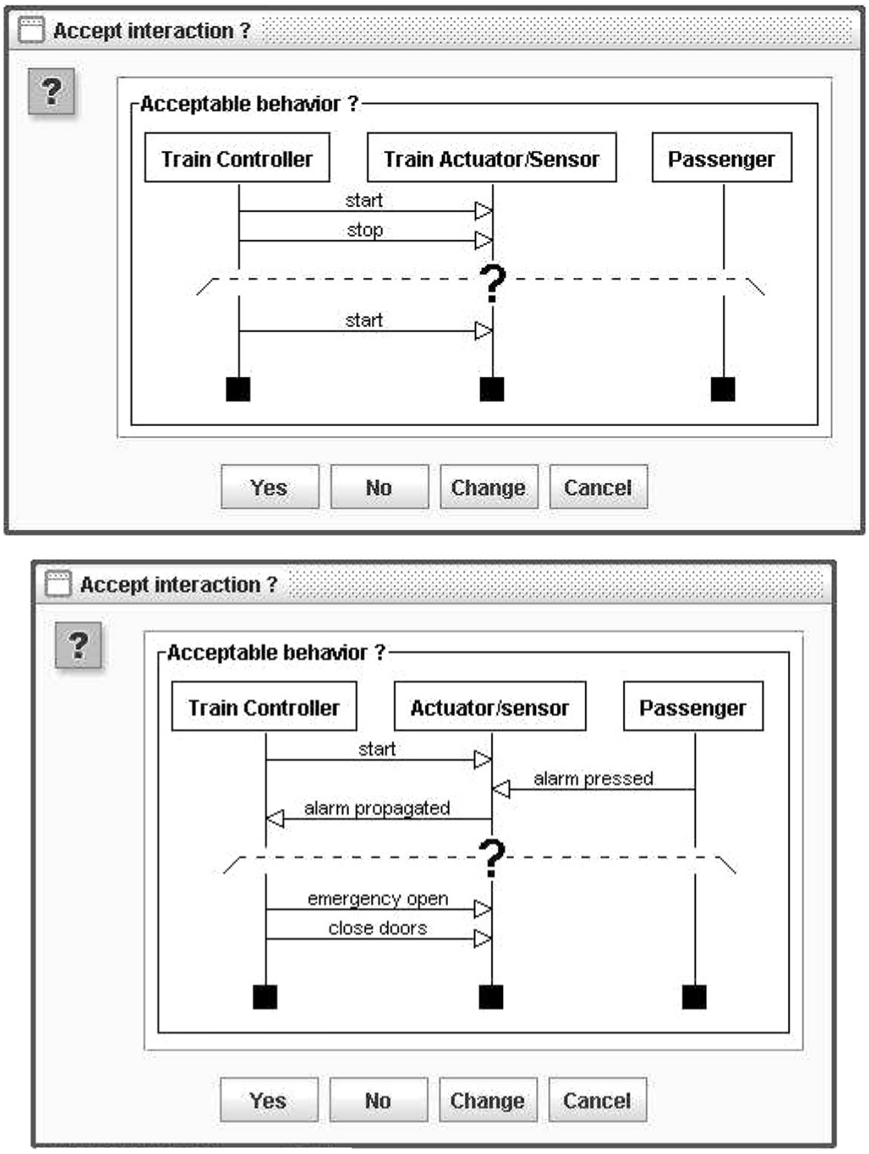

FIGURE 3 Typical scenario queries.

of states and a set of transitions between states. Each transition is labeled by an event. Strictly speaking, such machines do not model the agents themselves but how they interact between them. This is how system models are typically used by system analysts.

The state machines in this article are a particular class of automata called LTS (Magee and Kramer 1999). The QSM algorithm described in Section 4 induces a global LTS from end users scenarios. This LTS represents a global model of the acceptable behaviors. Simple techniques from automata theory (Hopcroft and Ullman 1979) can subsequently be used to build agent-specific LTS as described in Damas et al. (2005).

Definition 1 (Finite Automaton). *A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, δ is a transition function mapping $Q \times \Sigma$ to 2^Q, q_0 is the initial state, and F is a subset of Q identifying the accepting states. The automaton is called a DFA if for any q in Q and any e in Σ, $\delta(q, e)$ has at most one member.*

A string u is accepted by an automaton if there is a path from the initial state to some accepting state such that u is the concatenation of the transition symbols along this path. The language $L(A)$ accepted by an automaton A is the set of strings accepted by A. A LTS is an instance of DFA such that all states are accepting, that is, $Q = F$. This property is consistent with the fact that the strings accepted by a LTS are considered here as positive scenarios and that any prefix of a positive scenario must also be accepted.

One can question whether the finite state representation described here is fully adequate to model agent interactions. The first concern is the possibility of learning these models from scenarios simple enough to be proposed initially or classified interactively by end users. Fast induction algorithms are also required in order to guarantee a real-time response of this interactive learning procedure. Finite state modeling is also particularly adequate because of the existence of several automatic tools to analyze and check the induced models. Finally, they also offer a good starting point for automatic code generation. In a nutshell, even though finite-state modeling may seem crude, it is considered particularly convenient in the RE literature. Most of these models, however, are generally built by hand or semi-automatically with additional input information (see Section 7). The current article describes a novel application of learning techniques to construct these models fully automatically based on simple interactions with end users.

DFA INDUCTION

The reader familiar with DFA-induction techniques and the characterization of the RPNI algorithm can skip this section. Some of these notions are reviewed here to help newcomers to the field.

Quotient Automata and DFA Induction Search Space

Learning a language L aims at generalizing a positive sample S_+, possibly under the control of a negative sample S_-, with $S_+ \subseteq L$ and $S_- \subseteq \Sigma^* \backslash L$. When the induction technique produces a DFA, the learned language is regular. Any regular language L can be represented by its canonical automaton *A(L)*, that is, the DFA having the smallest number of states and accepting L. *A(L)* is unique up to a renumbering of its states (Hopcroft and Ullman 1979).

Generalizing a positive sample can be performed by merging states from an initial automaton that only accepts the positive sample. This initial automaton, denoted by $PTA(S_+)$, is called a prefix tree acceptor (PTA). It is the largest trimmed DFA accepting exactly S_+ (see Figure 4). The generalization operation is formally defined through the concept of quotient automaton.

Definition 2 (Quotient Automaton). *Given an automaton A and a partition π defined on its state set, the quotient automaton A/π is obtained by merging all states q belonging to the same partition subset $B(q, \pi)$. A state $B(q, \pi)$ in A/π thus corresponds to a subset of the states in A. A state $B(q, \pi)$ is accepting in A/π if and only if at least one state of $B(q, \pi)$ is accepting in A. Similarly, there is a transition on the letter a from state $B(q, \pi)$ to state $B(q', \pi)$ in A/π if and only if there is a transition on a from at least one state of $B(q, \pi)$ to at least one state of $B(q',\pi)$ in A.*

By construction of a quotient automaton, any accepting path in A is also an accepting path in A/π. It follows that for any partition π of the state set of A, $L(A/\pi) \supseteq L(A)$. In words, merging states in an automaton generalizes the language it accepts.

Learning a regular language is possible if S_+ is representative enough of the unknown language L and if the correct space of possible solutions is searched through. These notions are stated precisely hereafter.

Definition 3 (Structural Completeness). *A positive sample S_+ of a language L is structurally complete with respect to an automaton A accepting L if, when generating S_+ from A, every transition of A is used at least once and every final state is used as accepting state of at least one string.*

Rather than a requirement on the sample, structural completeness should be considered as a limit on the possible generalizations that are

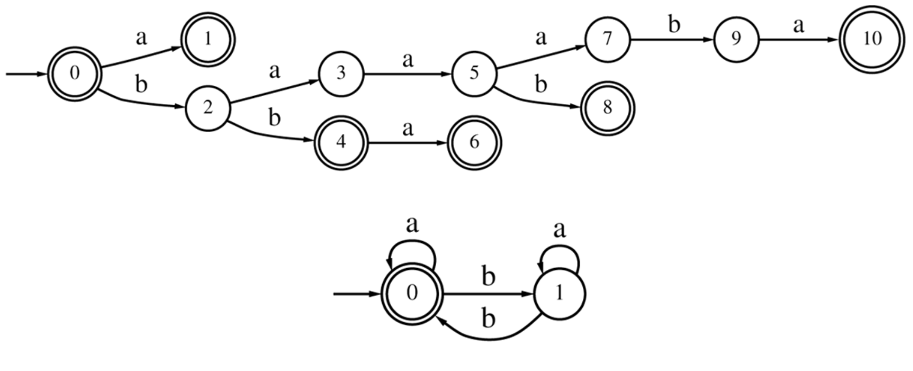

FIGURE 4 $PTA(S_+)$ (a) where $S_+ = \{\lambda, a, bb, bba, baab, baaaba\}$ is a structurally complete sample for the canonical automaton $A(L)$ (b). $A(L) = PTA(S_+)/\pi$ with $\pi = \{\{0, 1, 4, 6, 8, 9, 10\}, \{2, 3, 5, 7\}\}$.

allowed from a sample. If a proposed solution is an automaton in which some transition is never used while parsing the positive sample, no evidence supports the existence of this transition and this solution should be discarded.

Theorem 1 (DFA Search Space). *If a positive sample S_+ is structurally complete with respect to a canonical automaton A(L), then there exists a partition of the state set of PTA(S_+) such that PTA$(S_+)/\pi =$ A(L)* (*Dupont, Miclet,* and *Vidal* 1994).

This result defines the search space of the DFA induction problem as the set of all automata which can be obtained by merging states of the PTA. Some automata of this space are not deterministic but an efficient determinization process can enforce the solution to be a DFA (see Section 4).

Figure 4 presents the prefix tree acceptor built from the sample $S_+ = \{\lambda, a, bb, bba, baab, baaaba\}$, which is structurally complete with respect to the canonical automaton. This automaton is a quotient of the PTA for the partition $\pi = \{\{0,1,4,6,8,9,10\},\{2,3,5,7\}\}$ of its state set.

To summarize, learning a regular language L can be performed by identifying the canonical automaton *A(L)* of L from a positive sample S_+. If the sample is structurally complete with respect to this target automaton, it can be derived by merging states of the PTA built from S_+. A negative sample S_- is used to guide this search and avoid overgeneralization. In the sequel, $||S||$ denotes the sum of the lengths of the strings in a sample S.

The size[2] of this search space makes any trivial enumeration algorithm irrelevant for any practical purposes. Moreover, finding a minimal consistent DFA is a NP-complete problem (Gold 1978; Angluin 1978). Interestingly, only a fraction of this space is efficiently searched through by the RPNI algorithm or the QSM algorithm described in Section 4.

Characteristic Samples for the RPNI Algorithm

We do not fully detail the RPNI algorithm in the present section, but the original version forms a particular case of our interactive algorithm QSM, as discussed in Section 4. The convergence of RPNI to the correct automaton *A(L)* is guaranteed when the algorithm receives a sample as input that includes a characteristic sample of the target language (Oncina and García 1992). A proof of convergence, is presented in Oncina, García, and Vidal (1993) in the more general case of transducer learning. We review here the notion of a characteristic sample as the definition of relevant membership queries is related with this notion. Some additional definitions are required here.

Definition 4 (Short Prefixes and Suffixes). *Let Pr(L) denote the set of prefixes of L, with* $Pr(L) = \{u | \exists v, uv \in L\}$. *The right quotient of L by u, or set of suffixes of u in L, is defined by* $L/u = \{v | uv \in L\}$. *The set of short prefixes Sp(L) of L is defined by* $Sp(L) = \{x \in Pr(L) | \neg\exists u \in \Sigma^*$ *with* $L/u = L/x$ *and* $u < x\}$.

In a canonical automaton *A(L)* of a language *L*, the set of short prefixes is the set of the first strings in standard order[3] <, each of which leads to a particular state of the canonical automaton. Consequently, there are as many short prefixes as states in *A(L)*. In other words, the short prefixes uniquely identify the states of *A(L)*. The set of short prefixes of the canonical automaton of Figure 4 is $Sp(L) = \{\lambda, b\}$.

Definition 5 (Language Kernel). *The kernel N(L) of the language L is defined as* $N(L) = \{xa | x \in Sp(L), a \in \Sigma, xa \in Pr(L)\} \cup \{\lambda\}$.

The kernel is made of the short prefixes extended by one letter and the empty string. By construction $Sp(L) \subseteq N(L)$. The kernel elements represent the transitions of the canonical automaton *A(L)*, since they are obtained by adding one letter to the short prefixes that represent the states of *A(L)*. The kernel of the language defined by the canonical automaton of Figure 4 is $N(L) = \{\lambda, a, b, ba, bb\}$.

Definition 6 (Characteristic Sample). *A sample* $S^c = (S^c_+, S^c_-)$ *is characteristic for the language L and the algorithm RPNI if it satisfies the following conditions:*

1. $\forall x \in N(L)$, ***if*** $x \in L$ ***then*** $x \in S^c_+$ ***else*** $\exists u \in \Sigma^*$ *such that* $xu \in S^c_+$.
2. $\forall x \in Sp(L), \forall y \in N(L)$ ***if*** $L/x \neq L/y$ ***then*** $\exists u \in \Sigma^*$ *such that* $(xu \in S^c_+$ *and* $yu \in S^c_-)$ *or* $(xu \in S^c_-$ *and* $yu \in S^c_+)$.

Condition 1 guarantees that each element of the kernel belongs to S^c_+ if it also belongs to the language or, otherwise, is prefix of a string of S^c_+. One can easily check that this condition implies the structural completeness of the sample S^c_+ with respect to *A(L)*. In this case, Theorem 1 guarantees that the automaton *A(L)* can be derived by merging states from $PTA(S^c_+)$. When an element *x* of the short prefixes and an element *y* of the kernel do not have the same set of suffixes ($L/x \neq L/y$), they necessarily correspond to distinct states in the canonical automaton. In this case, condition 2 guarantees that a suffix *u* would distinguish them. In other words, the merging of a state corresponding to a short prefix *x* in $PTA(S^c_+)$ with another state corresponding to an element *y* of the kernel is made incompatible by the existence of *xu* in S^c_+ and *yu* in S^c_- or the converse.

To sum up, good examples to learn a canonical automaton *A(L)* guarantee to avoid merging of nonequivalent states q and q' (two states are equivalent if and only if they have the same set of suffixes in the target language). These good examples are the short prefixes of q and q', respectively, concatenated with the same suffix u to form a positive example from one state and a negative example from the other.

There may exist several distinct characteristic samples for a given language L as several suffixes u may satisfy condition 1 or 2. Note that if $|Q|$ denotes the number of states of the canonical automaton $A(L)$, the set of short prefixes contains $|Q|$ elements and the kernel has $\mathcal{O}(|Q| \cdot |\Sigma|)$ elements. Hence, the number of strings in a characteristic sample is given by

$$|S_+^c| = \mathcal{O}(|Q|^2 \cdot |\Sigma|) \quad \text{and} \quad |S_-^c| = \mathcal{O}(|Q|^2 \cdot |\Sigma|).$$

One can verify that $S = (S_+, S_-)$, with $S_+ = \{\lambda, a, bb, bba, baab, baaaba\}$ and $S_- = \{b, ab, aba\}$, forms a characteristic sample for the language accepted by the canonical automaton in Figure 4.

Note that the definition of a characteristic sample given above may be considered quite strong. It is, however, the standard definition of such a sample for the RPNI algorithm (Oncina and García 1992; Dupont 1996). It is based on a worst case analysis which does not make full use of the exact order in which state pairs are considered during the merging process. It does not rely on a specific order between the letters of the alphabet either. As observed in the experiments described in Section 6.2, a fraction of such a sample is often enough to observe very high generalization accuracy for randomly generated target DFAs. This observation is also consistent with the results reported in Lang et al. (1998).

QSM: AN INTERACTIVE STATE-MERGING ALGORITHM WITH MEMBERSHIP QUERIES

Algorithm 1 gives the pseudo-code of the QSM algorithm, a query driven state-merging DFA induction technique. The QSM algorithm takes a scenario collection as input and produces a consistent DFA as output. The completion of the initial scenario collection with classified scenarios that are generated during learning is another output of the algorithm. The input collection must contain at least one positive scenario. The generated DFA covers all positive scenarios in the final collection and excludes all negative ones.

The induction process can be described as follows: it starts by constructing an initial DFA covering all positive scenarios only. The induced DFA is then successively generalized under the control of the available negative scenarios and newly generated scenarios classified by the end user. This generalization is carried out by successively merging well-selected state pairs of the initial automaton. The induction process is such that, at any step, the current DFA covers all positive scenarios and excludes all negative ones, including the interactively classified ones. In the sequel, a DFA is said to be compatible with respect to a set of scenarios if it covers all positive scenarios in that set and excludes all negative ones. By extension, two states are said compatible for merging (resp. incompatible) if the quotient DFA which results from their merging is compatible (resp. incompatible) with the current set of scenarios.

```
Algorithm QSM
Input: A nonempty initial scenario collection $(S_+, S_-)$
Output: A DFA $A$ consistent with an extended collection $(S_+, S_-)$
$A \leftarrow$ Initialize$(S_+, S_-)$
while $(q, q') \leftarrow$ ChooseStatePairs$(A)$ do
    $A_{new} \leftarrow$ Merge$(A, q, q')$
    if Compatible$(A_{new}, S_+, S_-)$ then
        while $Query \leftarrow$ GenerateQuery$(A, A_{new})$ do
            if CheckWithEndUser$(Query)$ then
                $S_+ \leftarrow S_+ \cup Query$
            else
                $S_- \leftarrow S_- \cup Query$
                return QSM$(S_+, S_-)$
        $A \leftarrow A_{new}$
return $A, (S_+, S_-)$
```

ALGORITHM 1 QSM, an interactive state-merging algorithm with membership queries.

The `Initialize` function of QSM returns an initial candidate automaton built from S_+ and[4] S_-. Next, pairs of states are iteratively chosen from the current solution according to the `ChooseStatePairs` function.[5] The quotient automaton obtained by merging such states, and possibly some additional states, is computed by the `Merge` function. The compatibility of this quotient automaton with the learning sample is then checked by the `Compatible` function using available negative scenarios. When compatible, new scenarios are generated through the `GenerateQuery` function and submitted to the end user for classification (see Section 4.2). Scenarios classified as positive or negative are added to the initial collection with their respective labels. If all generated scenarios are classified as positive, the quotient automaton becomes the current candidate solution. The process is iterated until no more pair of states can be considered for merging. When a generated scenario is classified as negative, the algorithm is recursively called on the extended scenario collection.

The original RPNI algorithm can be seen as a particular instance of QSM when no query is generated or, equivalently, without the inner **while** loop. The advantage of QSM is that a finer control of the generalization offered by the state-merging operations can be obtained by validating these generalizations with an oracle. Section 4.1 describes the general process of merging compatible state pairs, while section 4.2 focuses on the generation of queries submitted to the end user. Section 4.3 discusses the adaptation of QSM to the blue-fringe strategy.

Merging Compatible State Pairs

The various functions that control how merging is performed from an initial automaton are described below.

Initialize The `Initialize` function returns the prefix tree acceptor built from S_+ and the proper prefixes from S_-. The PTA built from the initial scenario collection in Figure 5 is shown on top of Figure 6. According to the modeling hypothesis discussed in Section 2.2 (all states in a LTS are accepting states), all PTA states are labeled as accepting. This is a specificity of the application domain considered here, since any prefix of a positive scenario is also a positive scenario. Hence, a single positive scenario defines several positive examples. The algorithm proposed here would actually work for arbitrary positive samples not satisfying this property.

ChooseStatePairs The candidate solution is refined by merging well-selected state pairs. The `ChooseStatePairs` function determines which pairs to consider for such merging. It relies on the standard

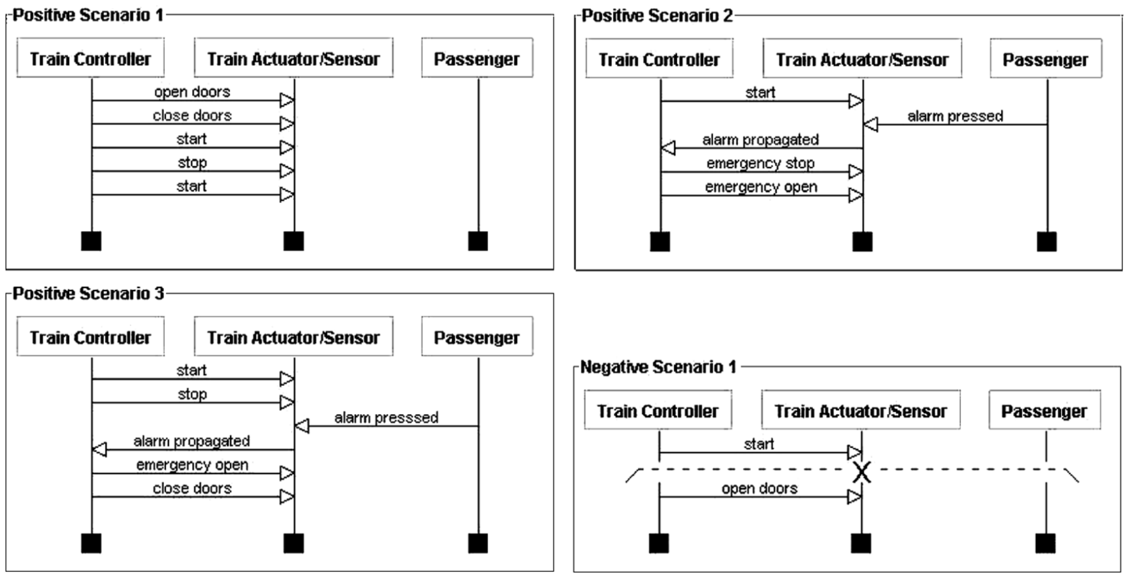

FIGURE 5 Initial positive and negative scenarios for a train system.

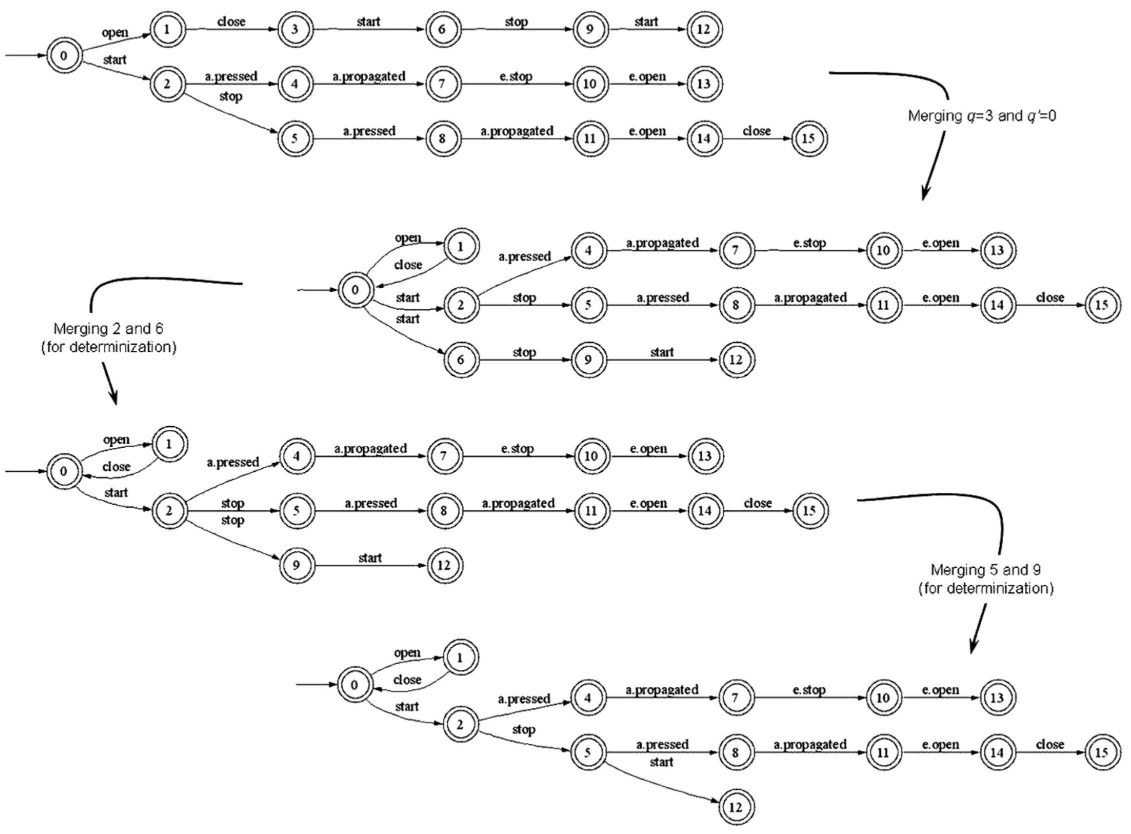

FIGURE 6 A typical induction step of the QSM algorithm.

order $<$ on strings. Each state of $PTA(S_+)$ can be labeled by its unique prefix from the initial state. Since prefixes can be sorted according to that order, the states can be ranked accordingly. For example, the PTA states in Figure 6 are labeled by their rank according to this order. The algorithm considers states q of $PTA(S_+)$ in increasing order. The state pairs considered for merging only involve such state q and any state q' of lower rank. The q' states are considered in increasing order as well. This particular ordering is specific to the original RPNI algorithm.

Merge The `Merge` function merges the two states (q, q') selected in order to compute a quotient automaton, that is, to generalize the current set of positive examples. In the example of Figure 6, we assume that states 0, 1, and 2 were previously determined not to be compatible for merging (through negative scenarios initially submitted or generated scenarios that were rejected by the user). Merging a candidate state pair may produce a nondeterministic automaton (NFA). For example, after having merged $q = 3$ and $q' = 0$ in the upper part of Figure 6, two transitions labeled `start` from state 0 lead to states 2 and 6, respectively. In such a case, the `Merge` function merges states 2 and 6, and recursively, any further pair of states that introduces nondeterminism.

We call merging for determinization this recursive operation of removing nondeterminism. This operation guarantees that the current solution at any step is a DFA. It produces an automaton that may accept a more general language than the NFA it starts from, and as such, it is not equivalent to the standard algorithm to transform a NFA into a DFA accepting the same language (Hopcroft and Ullman 1979). The time complexity of merging for determinization is a linear function of the number of states of the NFA it starts from. When two states are merged, the rank of the resulting state is defined as the lowest rank of the pair; in particular, the rank of the merged state when merging q and q' is defined as the rank of q' by construction. If no compatible merging can be found between q and any of its predecessor states according to $<$, state q is said to be consolidated (in the example, states 0, 1, and 2 are consolidated).

Compatible The `Compatible` function checks whether the automaton A_{new} correctly rejects all negative scenarios. As seen in Algorithm 1, the quotient automaton is discarded by QSM when it is detected not to be compatible with the negative sample.

Generating Queries Submitted to the End User

This section describes how membership queries are generated in the QSM algorithm and how the answers provided by the end user are processed. A complexity analysis of this algorithm is provided afterwards.

GenerateQuery When an intermediate solution is compatible with the available scenarios, new scenarios are generated for classification by the end user as positive or negative. The aim is to avoid poor generalizations of the learned language. The notion of characteristic sample drives the identification of which new scenarios should be generated as queries. Recall from Section 3.2 that a sample is characteristic of a language *L*, called here the target language, if it contains enough positive and negative information. On the one hand, the required positive information is the set of short prefixes *Sp(L)*, which form the shortest histories leading to each target DFA state. This positive information must also include all elements of the kernel *N(L)* which represents all system transitions, that is, all shortest histories followed by any admissible event. If such positive information is available, the target machine can always be derived from the PTA by an appropriate set of merging operations. On the other hand, the negative scenarios provide the necessary information to make incompatible the merging of states which should be kept distinct. A negative scenario that excludes the merging of a state pair (q, q') can be simply made of the shortest history leading to q' followed by any suffix, i.e., any valid continuation from state q as detailed below.

Consider the current solution of our induction algorithm when a pair of states (q, q') is selected for merging. By construction, q' is always a consolidated state at this step of the algorithm, that is, q' is considered to be in $Sp(L)$. State q is always both the root of a tree and the child of a consolidated state. In other words, q is situated at one letter of a consolidated state, that is, q is considered to be in $N(L)$. States q and q' are compatible according to the available negative scenarios; they would be merged by the standard RPNI algorithm. In our extension, the tool will first confirm or infirm the compatibility of q and q' by generating scenarios to be classified by the end user. The generated scenarios are constructed as follows.

Let A denote the current solution, $L(A)$ the language generated by A, and A_{new} the quotient automaton computed by the `Merge` function at some given step. Let $x \in Sp(L)$ and $y \in N(L)$ denote the short prefixes of q' and q in A, respectively. Let $u \in L(A)/y$ denote a suffix of q in A. A generated scenario is a string xu such that $xu \in L(A_{new}) \backslash L(A)$. This string can be further decomposed as xvw such that $xv \in L(A)$. A generated scenario xu is thus constructed as the short prefix of q' concatenated with a suffix of q in the current solution, provided the entire behavior is not yet accepted by A. Such scenario is made of two parts: the first part xv is an already accepted behavior, whereas the second part w provides a continuation to be checked for acceptance by the end user. When submitted to the end user, the generated scenario can always be rephrased as a question: after having executed the first episode (xv), can the system continue with the second episode (w)? Consider the example in Figure 6 with selected state pair $q = 3$, $q' = 0$. As q' is the root of the PTA, its short prefix is the empty string. The suffixes of q here yield one generated question (Figure 7), which can be rephrased as follows: when having started and stopped the train, can the controller restart it? One can see that the first episode of this scenario in Figure 6 is already accepted by A, whereas the entire behavior is accepted in A_{new}. The suffixes selected by our tool for generating queries are always the entire branches of the tree rooted at q. The aim is to help the end user to determine more easily whether the generated scenario should be rejected. The boundary between the first (xv) and second (w) episodes of this scenario can be easily determined by comparing A and A_{new} as a side product of the merging for determinization implemented in the `Merge` function.

Check WithEndUser When a new scenario is generated, it is submitted as a membership query to the end user. If the end user classifies the *Query* as positive, it is added to the collection of positive scenarios. This addition

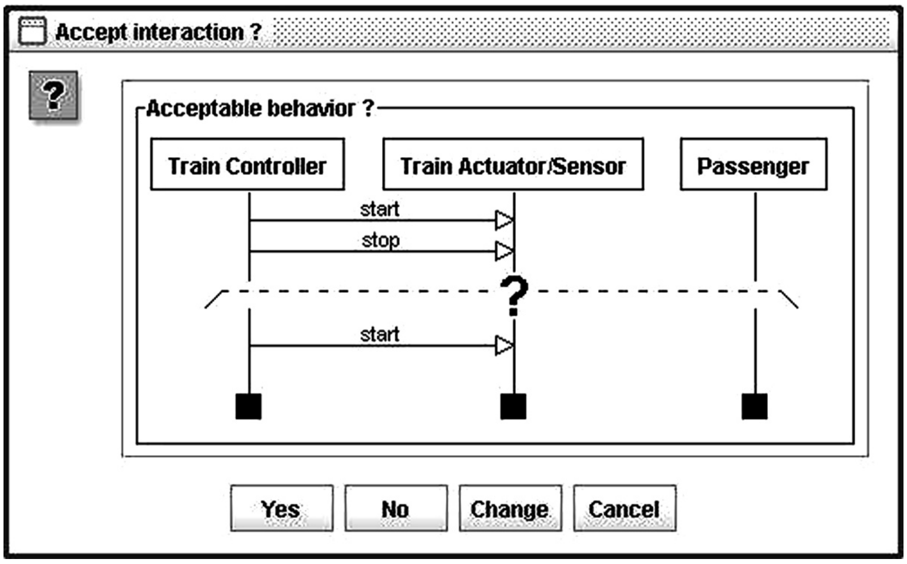

FIGURE 7 A new scenario to be classified by the end user.

changes the search space as it extends S_+ and consequently $PTA(S_+)$. However, this extension is implicit as the new solution A_{new} is, by construction, also a quotient automaton of this extended PTA. When the *Query* is classified as negative the induction process is recursively started[6] on the extended scenario collection.

The QSM algorithm has a polynomial time complexity in the size of the learning sample. An upper bound on the time complexity can be derived as follows.

Let $n \Box \mathcal{O}(||\mathcal{S}_+|| + ||\mathcal{S}_-||)$ denote the number of states of the PTA built from the initial collection of scenarios. For a fixed collection of scenarios, there are $\mathcal{O}(n^2)$ state pairs which are considered for merging. The `Merge` and `Compatible` functions have a time complexity linear in n. The `GenerateQuery` is a side product of the `Merge` function and does not change its complexity. The function `CheckWithEndUser` is assumed to run in constant time. Hence, for a fixed scenario collection, the time complexity is the same as for the RPNI algorithm and is upper bounded by $\mathcal{O}(n^3)$. This bound is obviously not very tight. It assumes that all pairs of states considered by `ChooseStatePairs` appears to be incompatible, which is a very pessimistic assumption. Practical experiments often show that the actual complexity is much closer to the lower bound $\Omega(n)$.

The global complexity of QSM depends on the number of recursive calls, that is, the number of times a new scenario submitted to the end user is classified as negative. The way new scenarios are generated by the `GenerateQuery` function guarantees that the PTA built from the extended scenario collection has at most $\mathcal{O}(n^2)$ states. During the whole incremental learning process, there is at most one query for each transition in this tree. Consequently, the number of queries is bounded by $\mathcal{O}(n^2)$.

When QSM received a characteristic sample in the initial scenario collection (or any scenario collection considered when calling it recursively), it is guaranteed that no additional scenario can be classified as negative. It follows that QSM will no longer be called recursively and stops by returning the target model. Note that the size of such a characteristic sample is not necessarily reduced by the fact that any prefix of a positive scenario is also a positive scenario, since the number of negative examples it must contain is not affected by this property. An experimental study of the actual sample size required to observe the convergence of QSM and the number of queries submitted to the end user is detailed in Section 6.2.

Reducing the Number of Queries with Blue-Fringe

The order in which states are considered for merging by the `ChooseStatePairs` function described in Section 4.1 follows from the implicit assumption that the current sample is characteristic. Consequently, two states are considered compatible for merging if there is no suffix to distinguish among them. This can lead to a significant number of scenarios being generated to the end user, to avoid poor generalizations, when the initial sample is sparse and actually not characteristic for the target global LTS. To overcome this problem, our tool implements an optimized strategy known as Blue-Fringe (Lang et al. 1998). The difference lies in the way state pairs are considered for merging. The general idea is to detect early incompatible state pairs and, subsequently, first consider state pairs for which compatibility has the highest chance to be confirmed by the user through positive classification. The resulting "please confirm" interaction may also appear more appealing to the user.

Figure 8 gives a typical example of a temporary solution produced by the original algorithm. Three state classes can be distinguished in this DFA. The red states are the consolidated ones (0, 1, and 2 in this example).

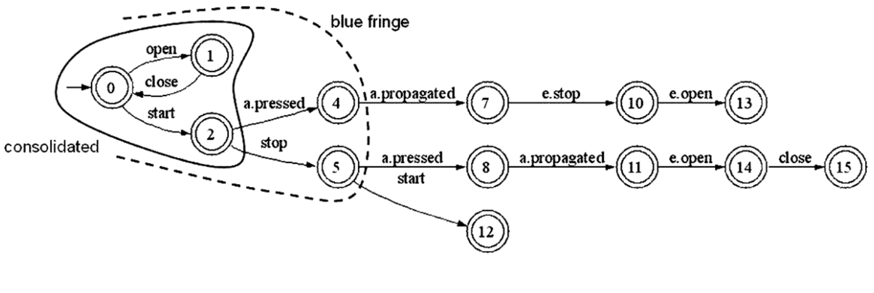

FIGURE 8 Consolidated states (red) and states on the fringe (blue) in a temporary solution.

Outgoing transitions from red states lead to blue states unless the latter have already been labeled as red. Blue states (4 and 5 in this case) form the blue fringe. All other states are white states.

The original `ChooseStatePairs` function described in Section 4.1 considers the lowest-rank blue state first (state 4 here) for merge with the lowest-rank red state (0). When this choice leads to a compatible quotient automaton, generated scenarios are submitted to the end user (in this case, a scenario equivalent to the string `alarm propagated`, `emergency stop`, `emergency open`). The above strategy may lead to multiple queries being generated to avoid poor generalization. Moreover, such queries may be nonintuitive for the user, e.g., the `alarm propagated` event is sent to the train controller without having been fired by the `alarm pressed` event to the sensor.

To select a state pair for merging, the blue-fringe strategy evaluates all (red, blue) state pairs first. The `ChooseStatePairs` function now calls the `Merge` and `Compatible` functions before selecting the next state pair. If a blue state is found to be incompatible with all current red states, it is immediately promoted to red; the blue fringe is updated accordingly and the process of evaluating all (red, blue) pairs is iterated. When no blue state is found to be incompatible with red states, the most compatible (red, blue) pair is selected for merging. Note that the `Initialize` function now returns an augmented prefix tree acceptor $PTA(S_+, S_-)$. It stores the prefixes of all positive and negative strings, with accepting states being labeled either as positive or negative. The `Compatible` function now returns a compatibility score instead of a boolean value. The score is defined as $-\infty$, when in the merging process for determinization, merging the current (red, blue) pair requires some positive accepting state to be merged with some negative accepting state; this score indicates an incompatible merging. Otherwise, the compatibility score measures how many accepting states in this process share the same label (either $+$ or $-$). The (red, blue) pair with the highest compatibility score is considered first. The above strategy can be further refined with a compatibility threshold α as additional input parameter. Two states are considered to be compatible if their compatibility score is above that threshold. This additional parameter controls the level of generalization, since increasing α decreases the number of state pairs that are considered compatible for merging; it thus decreases the number of generated queries.

On the simple train example of this article, the QSM algorithm with the original RPNI state-merging order learns the global LTS correctly by submitting 20 scenarios to the end user (17 should be rejected and only 3 should be accepted). With the blue-fringe strategy, the same LTS is synthesized with only three scenarios being submitted (one to be rejected and two to be accepted). Further comparative results are detailed in Section 6.

INTEGRATION OF OPTIONAL DOMAIN KNOWLEDGE

The interactive QSM algorithm described in Section 4 always provides a DFA consistent with the available positive and negative scenarios. The blue-fringe strategy can also be applied to reduce the number of additional scenarios submitted to the end user. This strategy relies on two equivalence classes partitioning the states of an augmented PTA. These classes correspond to the positive and negative accepting states respectively.[7] All states belonging to the same class are not necessarily merged in the final solution but the `Compatible` function guarantees that only states belonging to the same class can be merged.

This approach can be extended to incorporate various sources of domain knowledge. This knowledge refines the equivalence partition and further constrains the compatible merging operations. From an algorithmic point of view, inclusion of domain knowledge is thus straightforward but the resulting approach both speeds up the search and guarantees that the proposed solution is consistent with the domain knowledge.

Let us stress that domain knowledge is optional here as the QSM algorithm can work without it. However, if available, domain knowledge helps to further reduce the number of queries submitted to the end user. The next sections describe how to include fluents, models of external system components, descriptive statements, and goals in the induction process.

Propagating Fluents

The notion of fluent has been introduced in Giannakopoulou and Magee (2003).

Definition 7 (Fluent). *A* fluent *is a proposition defined by a set $Init_{Fl}$ of initiating events, a set $Term_{Fl}$ of terminating events, with $Init_{Fl} \cap Term_{Fl} = \emptyset$, and an initial boolean value.*

For example, the fluent *DoorsClosed* describes states of the train doors as being either `closed` (*DoorsClosed* = *true*) or `open` (*DoorsClosed* = *false*), and describes which event is responsible for which state change:

$$DoorsClosed = \langle\{\texttt{close doors}\}, \{\texttt{open doors}, \texttt{emergency open}\}\rangle \quad \text{initially } true.$$

The value of every fluent can be computed on each PTA state by symbolic execution, starting from the initial state associated with the initial value for each fluent. The PTA states are then decorated with the conjunction of such values. Two states in $PTA(S_+, S_-)$ belong to the same equivalence class if they have the same value for every fluent. The decoration of the merged states is simply inherited from the states being merged.

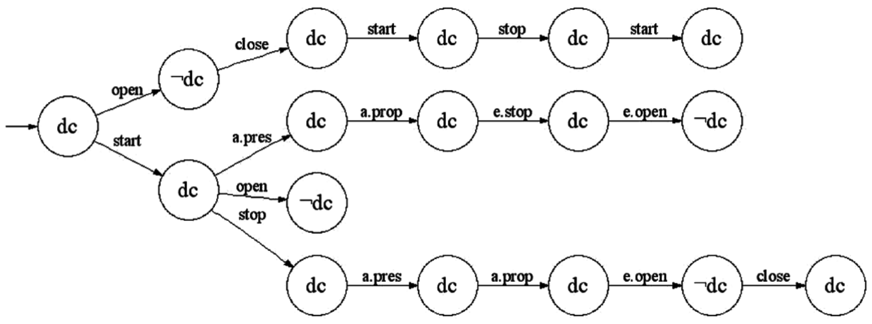

FIGURE 9 Propagating fluents. dc is a shorthand for *DoorsClosed*; `a.pres`, `a.prop`, `e.open` stand for `alarm pressed`, `alarm propagated`, `emergency open`, respectively.

Figure 9 shows the result of propagating the values of the fluent *DoorsClosed* along the augmented PTA built from the scenarios described in Figure 5.

Unfolding Models of External Components

Quite often, the components being modeled need to interact with other components in their environment, e.g. legacy components in a bigger existing system, foreign components in an open system, etc. In such cases, the behavior of external components is generally known—typically through some behavioral model (Hall and Zisman 2004). Here we assume that external components are known by their LTS model. For example, Figure 10 shows the LTS for a legacy alarm sensor in our train system. When the alarm button is pressed by a passenger, this component propagates a corresponding signal to the train controller.

The PTA states can also be decorated with state labels from this external LTS by unfolding the latter on the PTA. Such decoration is performed by jointly visiting the PTA and the external LTS. The latter synchronizes on shared events and stays in its current state on other events. Figure 11 shows the result of unfolding the alarm sensor LTS from Figure 10 on the augmented PTA built from the scenarios described in Figure 5. Each state in Figure 11 is labeled with the number of the corresponding state in the alarm sensor LTS. Two states belong to the same equivalence class if they have the same external LTS state label.

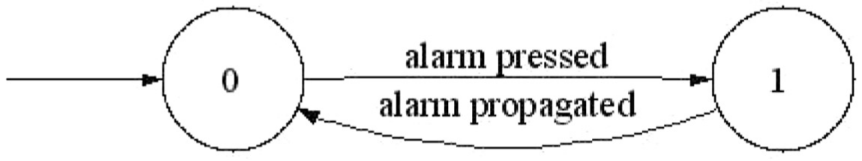

FIGURE 10 Alarm sensor LTS.

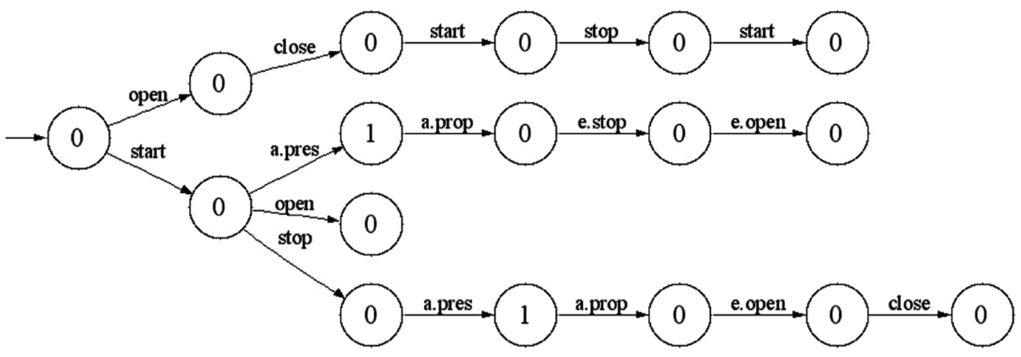

FIGURE 11 Unfolding the alarm sensor model.

Injecting Descriptive Statements and Goals in the Induction Process

Descriptive statements about the domain, called domain properties, or prescriptive statements of intent about the target system, called goals, can be expressed declaratively with fluents in linear temporal logic (LTL) (Giannakopoulou and Magee 2003). Linear temporal logic assertions use standard operators for temporal referencing such as:

$$\square(\text{always in the future}), \diamond\ (\text{some time in the future}),$$
$$\rightarrow (\text{implies in the current state}), \Rightarrow (\text{always implies}).$$

For example, in an extended version of our running example, the statement

$$\square(\mathit{HighSpeed} \rightarrow \mathit{Moving})$$

expresses a physical law stating that all scenarios for which a train is running at high speed while not moving should be considered as negative. A goal requiring train doors to remain closed while the train is moving is formalized as

$$\mathit{DoorsClosedWhileMoving} = \square(\mathit{Moving} \rightarrow \mathit{DoorsClosed}).$$

We restrict here our attention to statements that can be formalized as LTL safety properties. For such properties, a tester can be automatically generated (Giannakopoulou and Magee 2003). A tester for a property is a LTS extended with a negative accepting state such that every path leading to this state violates the property. The tester LTS for the goal *DoorsClosed-WhileMoving* is represented in Figure 12 (the black state is the negative accepting state). Any event sequence leading to the black state from the

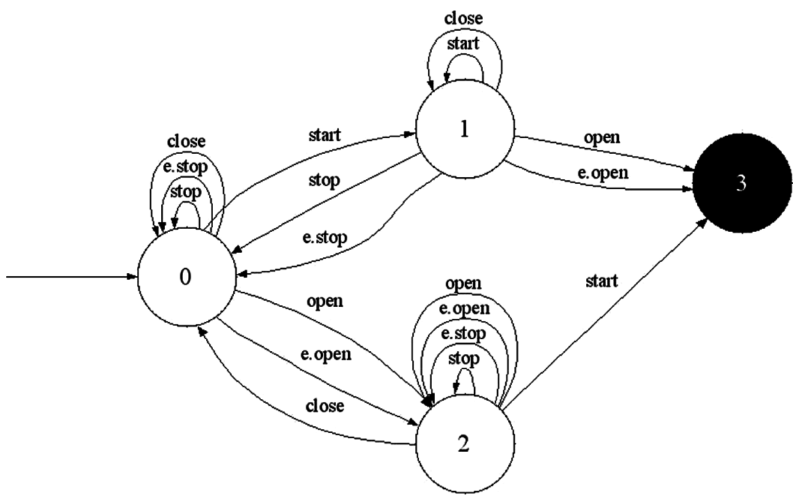

FIGURE 12 Tester LTS for the goal *DoorsClosedWhileMoving*. `e.stop`, `e.open` stand for `emergency stop`, `emergency open`, respectively.

initial state corresponds to an undesired system behavior. In particular, the event sequence `start`, `open` corresponds to the initial negative scenario in our running example (see Figure 5). The tester of Figure 12 provides many more negative scenarios, actually an infinite number of negative scenarios due to the cyclic nature of such LTS. The property tester is used to constrain the induction process in the same way as an external component LTS. The PTA and the tester are traversed jointly in order to decorate each PTA state with the corresponding tester state. Two states belong in the same equivalence class if they have the same property tester state. This technique has the additional benefit of ensuring that the synthesized global LTS satisfies the considered goal or domain property.

PRACTICAL EVALUATIONS

We compare here how the techniques described in Sections 4 and 5 perform in practice. Section 6.1 describes experiments on three typical (in the RE literature) case studies of varying complexity. Section 6.2 addresses specifically the question of how the QSM algorithm scales with the size of target machines. Since standard RE test cases are relatively small, these additional experiments are performed on synthetic data without domain knowledge.

Requirement Engineering Case Studies

The first case study is a mine pump system inspired from Joseph (1996). The second is an extended version of the train system used here as a running example. The third is a phone system handling communications between a caller and a callee.

The first objective is to measure the impact of the blue-fringe strategy versus the original RPNI state merging order. The second objective is to assess the impact of constraining induction through fluents, models of external components, domain descriptions, and goals. Impact is measured in terms of the number of queries and the adequacy of the induced models, as estimated by a software analyst.

For each case study, we proceed in two steps:

1. (a) Design a scenario collection allowing for meaningful subsequent comparison, that is, a scenario collection sufficiently rich to allow an adequate global LTS to be induced under one setting of the experiment at least.
   (b) Define a common set of fluent definitions identifiable from this scenario collection.
2. Evaluate the techniques on this scenario collection, without and with fluents, goals, domain descriptions, or models of external components.

Condition (1a) amounts to require the scenario collection to be structurally complete. We used the ISIS tool itself to incrementally set up such a scenario collection. We started from an initial set of scenarios that end users would typically provide. By generating scenario queries, adding domain properties, and validating induced LTSs, we found a number of additional scenarios that were missing. These scenarios are added to the collection for the comparisons in step 2.

The size of the scenario collection resulting from step 1 is shown in Table 1. SC_+ and SC_- correspond to the number of positive and negative scenarios, respectively. A single positive scenario defines several positive examples as any prefix of a positive scenario is also positive. Similarly, a single negative scenario defines positive examples (all proper prefixes) and a single negative example (the full string). The average scenario length is reported in the last column. The sizes of the target global LTS in terms of the number of different event labels (alphabet size), states, and transitions are also reported. To perform the comparisons, an oracle was implemented to simulate the end user. This oracle knows the target LTS for each

TABLE 1 Case Studies

Problem	Events	States	Transitions	SC_+	SC_-	Avg. length
Mine Pump	8	10	13	3	0	8
Train	13	17	23	3	0	9
Phone	16	23	33	6	4	11

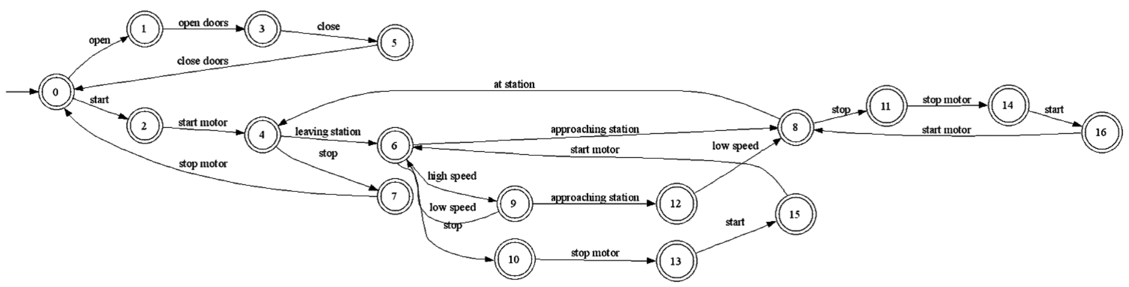

FIGURE 13 Train system case study.

problem and correctly classifies scenario queries as positive or negative. Figure 13 presents the target LTS for the complete train system.

RPNI Search Order Versus Blue-Fringe Strategy

Table 2 shows the number of queries the oracle had to answer and the adequacy of the induced model, in the three case studies, when no additional knowledge is used to constrain induction. Q_+ and Q_- denote the number of accepted and rejected scenario queries, respectively. The number of rejected scenario queries is drastically reduced thanks to the blue-fringe search strategy. Finally, the number of rejected scenarios tends to be much larger on these test cases than the number of accepted ones. This observation confirms the usefulness of scenario queries. Negative answers force the induction algorithm to be restarted when an incorrect search path has been taken.

The superiority of blue-fringe over RPNI had already been observed in a noninteractive setting (Lang et al. 1998) when the learning sample is not characteristic. Interestingly, the blue-fringe strategy also pays off in our QSM interactive learning algorithm by reducing the number of membership queries to be submitted to an oracle. As blue-fringe is seen to be far superior to the RPNI strategy, subsequent comparisons are made only with blue-fringe.

Impact of Fluent Propagation

Table 3 shows the influence of fluent decorations to constrain the induction process. Note that the number of rejected scenario queries is

TABLE 2 RPNI Search Order Versus Blue-Fringe Strategy for the QSM Algorithm

Problem	Search	Q_+	Q_-	Model Adequacy
Mine Pump	RPNI	1	30	Missing/unauthorized paths
	Blue-Fringe	1	4	target found
Train	RPNI	4	83	target found
	Blue-Fringe	5	5	target found
Phone	RPNI	5	172	Missing/unauthorized paths
	Blue-Fringe	6	17	target found

TABLE 3 Impact of Fluent Propagation

Problem	Fluents	Q_+	Q_-	Model Adequacy
Mine Pump	0	1	4	target found
	1	1	1	target found
	2	1	0	target found
	3	1	0	target found
Train	0	5	5	target found
	1	5	3	target found
	2	5	3	target found
	3	5	3	target found
	4	5	2	target found
	5	5	0	target found
Phone	0	6	17	target found
	1	6	11	target found
	2	6	6	target found
	3	6	5	target found

decreasing with the number of fluents. For the same induced model, the number of accepted scenarios remains the same. Fluent-based state information can only increase the number of incompatible states.

Combined Results

Table 4 shows the results of a blue-fringe induction constrained with all available fluents, goals, and domain properties,[8] and foreign component(s) in each case study. A typical goal for the train system states that the train may never run at high speed when it comes near a station. For each problem, the first line corresponds to the simplest approach, that is, the RPNI search order and no domain knowledge. Goals and domain properties alone were observed as powerful as fluents to reduce the number of queries.

Real-Time Execution

Our interactive blue-fringe implementation has been tested for performance evaluation on an Intel Pentium IV, 1.8 GHz, 512 Mb with Java 5.0. These tests illustrate that the maximum time between two queries was 40 ms for the bigger case study. Hence, the interactions with the end

TABLE 4 Global Results of QSM on the RE Test Cases

Problem	Search	Fluents	Goals	External LTS	Q_+	Q_-	Model Adequacy
Mine Pump	RPNI	0	0	0	1	30	missing/unauthorized paths
	Blue-fringe	3	3	2	1	0	target found
Train	RPNI	0	0	0	4	83	target found
	Blue-Fringe	5	5	2	5	0	target found
Phone	RPNI	0	0	0	5	172	missing/unauthorized paths
	Blue-Fringe	3	3	1	7	3	target found

user are performed in real-time. We do not expect any performance problem with respect to the user interactivity for typical sizes of requirement models.

Experiments on Synthetic Datasets

The QSM algorithm is evaluated here on synthetic data in order to study its performance when the problem size (in terms of the number of states of the target machine) grows significantly beyond those of the standard RE test cases presented in Section 6.1. In this particular setting, we do not rely on additional domain knowledge. Section 6.2.1 describes the methodology used to generate automata, learning, and test samples. Section 6.2.2 discusses the gain in terms of generalization accuracy of the QSM algorithm with respect to the original RPNI or blue-fringe algorithms. The number of generated queries by QSM is reported in Section 6.2.3. Section 6.2.4 gives comparative performances in terms of induction CPU time.

Generation of Target Models, Training, and Test Data

The procedure described here is inspired from the Abbadingo competition (Lang et al. 1998). Experiments are made on automata of increasing sizes: $n = 20$, 50, 100, and 200 states with an alphabet of two letters. Randomly generated automata are trimmed to remove unreachable states and minimized to obtain canonical target machines. Moreover, only automata without sink state are kept for the experiments. Such states represent deadlocks in multi-agent systems considered in the RE context and should always be avoided. The number of states of a DFA generated using this procedure is approximately 3/5 of the requested size. The latter has been increased accordingly.

A sample composed of n^2 different strings has been initially synthesized by a random walk of the automaton. These strings have been generated using a uniform length distribution $[0, p+5]$, where p is the depth of the automaton. The random walk procedure is implemented to provide positive and negative strings in equal proportion.

A maximal sample size of $n^2/2$ strings was experimentally observed as offering the convergence for all tested algorithms. The learning experiments are launched on increasing proportions of this nominal training sample, i.e., 3%, 6%, 12.5%, 25%, 50%, and 100%. Test samples of at most $n^2/2$ strings are used to measure the generalization accuracy of the learned model. Training and test samples are guaranteed not to overlap. Moreover, in the case of the QSM algorithm, test samples do not contain the additional strings which were submitted to the oracle during the interactive learning phase.

An automatic oracle has been implemented to answer the questions asked during the execution of the QSM algorithm. This oracle correctly answers the membership queries since it has access to the target automaton. Experiments are performed on 10 randomly generated automata for each size and five randomly generated samples for each of them.

Generalization Accuracy

Figure 14 reports for several target sizes the proportion of independent test samples correctly classified while increasing the learning sample size. Comparative performances are given for RPNI, Blue Fringe, QSM-RPNI (QSM with the RPNI merging order), and QSM-Blue Fringe (QSM with the Blue Fringe strategy).

Known results are confirmed here since Blue Fringe outperforms RPNI for sparse training samples. Moreover, significant improvements of the generalization accuracy is observed thanks to the oracle. The QSM algorithm outperforms the original RPNI and Blue Fringe systematically. Interestingly, QSM-RPNI also overcomes the original Blue Fringe algorithm. When the classification rate of test samples reached 100%, the proposed solutions were always isomorphic to the target machines.

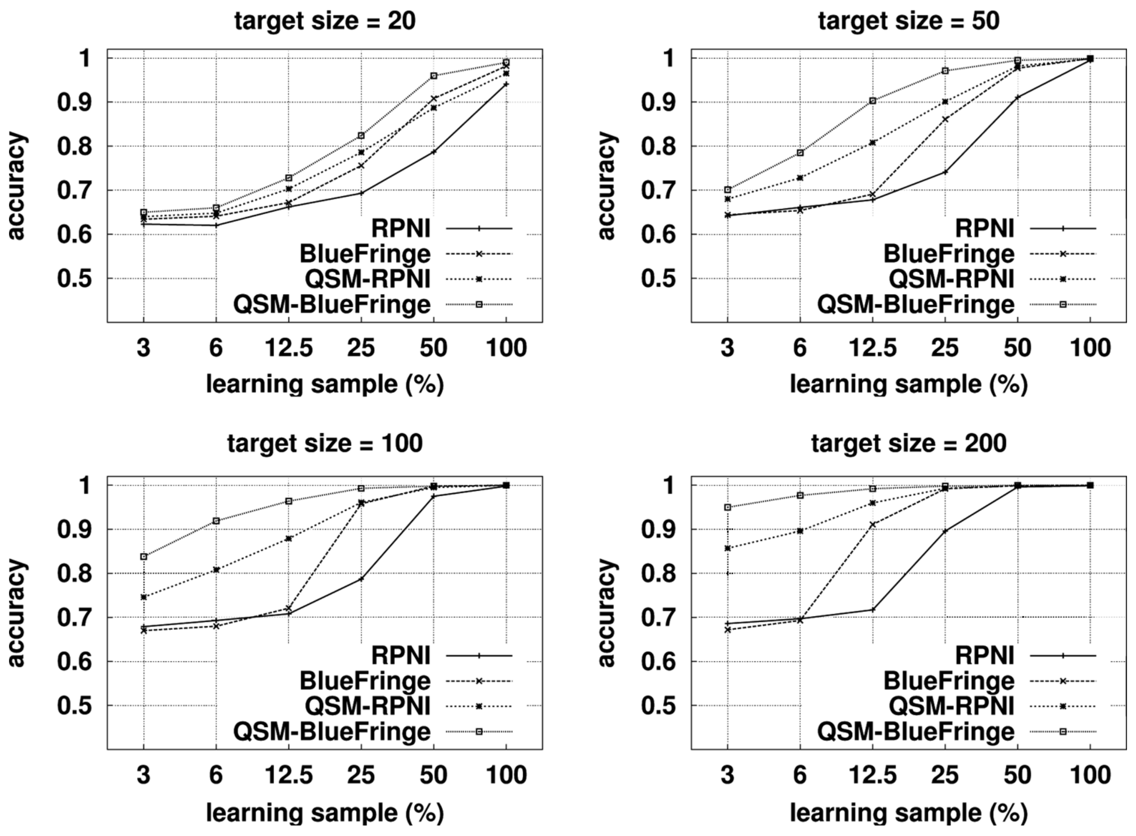

FIGURE 14 Classification accuracy.

The relative performances described above do not depend much on the target size. It should be noted that the learning sample sizes are well chosen to illustrate the convergence of the standard RPNI and Blue Fringe algorithms. When the target size grows (especially for $n = 200$) the QSM-Blue Fringe has already nearly converged with only 1.5% of the nominal learning sample size. Such a gain can also be explained by the role played by the oracle. The QSM algorithm tends to elicit a characteristic sample by submitting queries to the user. This issue is further investigated in Section 6.2.3, where the number of queries is reported as function of the learning sample size.

Number of Queries

An important aspect of QSM is the number of queries submitted to the oracle. When the oracle is an end user, as in the RE context, this factor indeed drives the usability of the approach in practice. Figure 15 presents the number of generated queries depending on the learning sample size for several target sizes. Results are given for QSM-RPNI and QSM-Blue Fringe. In each case, the number of generated strings that are classified by the oracle either as positive or negative are reported separately.

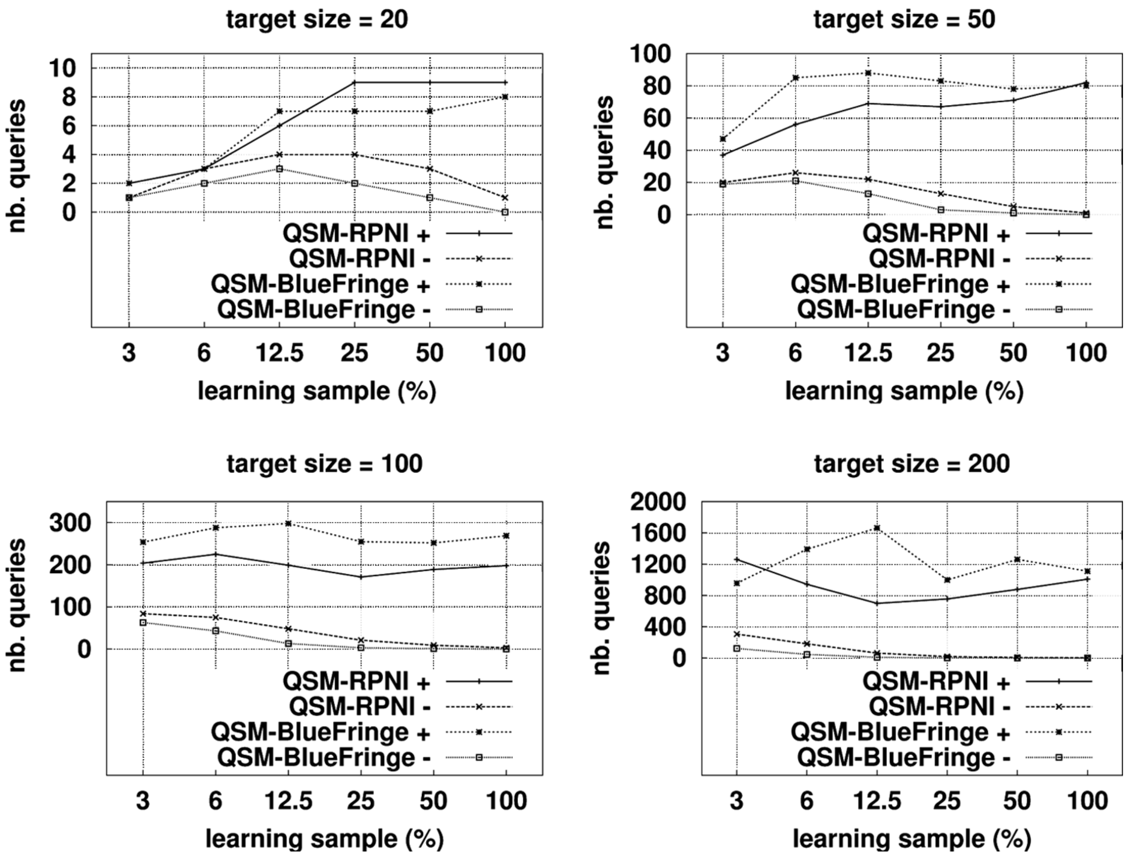

FIGURE 15 Number of queries.

The number of strings classified as positive tends to increase initially with the learning sample size before staying roughly constant when the learning process has nearly converged. The additional information required to guarantee correct identification depends mostly on the negative strings.

Figure 15 shows that QSM-Blue Fringe tends to generate fewer strings classified as negative by the oracle as compared to QSM-RPNI. The figure also shows that the number of negative strings decreases with the learning sample size. In other words, the number of strings classified as negative decreases while the algorithm converges. This illustrates that the generalization obtained while merging states is more sound when performed with more data.

Induction Time

Figure 16 reports the induction time while varying learning sample size and target size. All tests on synthetic data were executed with Java 5.0 on an Intel Pentium-IV 3 GHz computer with 1 Gb of RAM.

The RPNI and Blue Fringe algorithms go through different phases according to the amount of available data. Initially, the CPU time tends to increase with the learning sample size. In this first phase, the learning

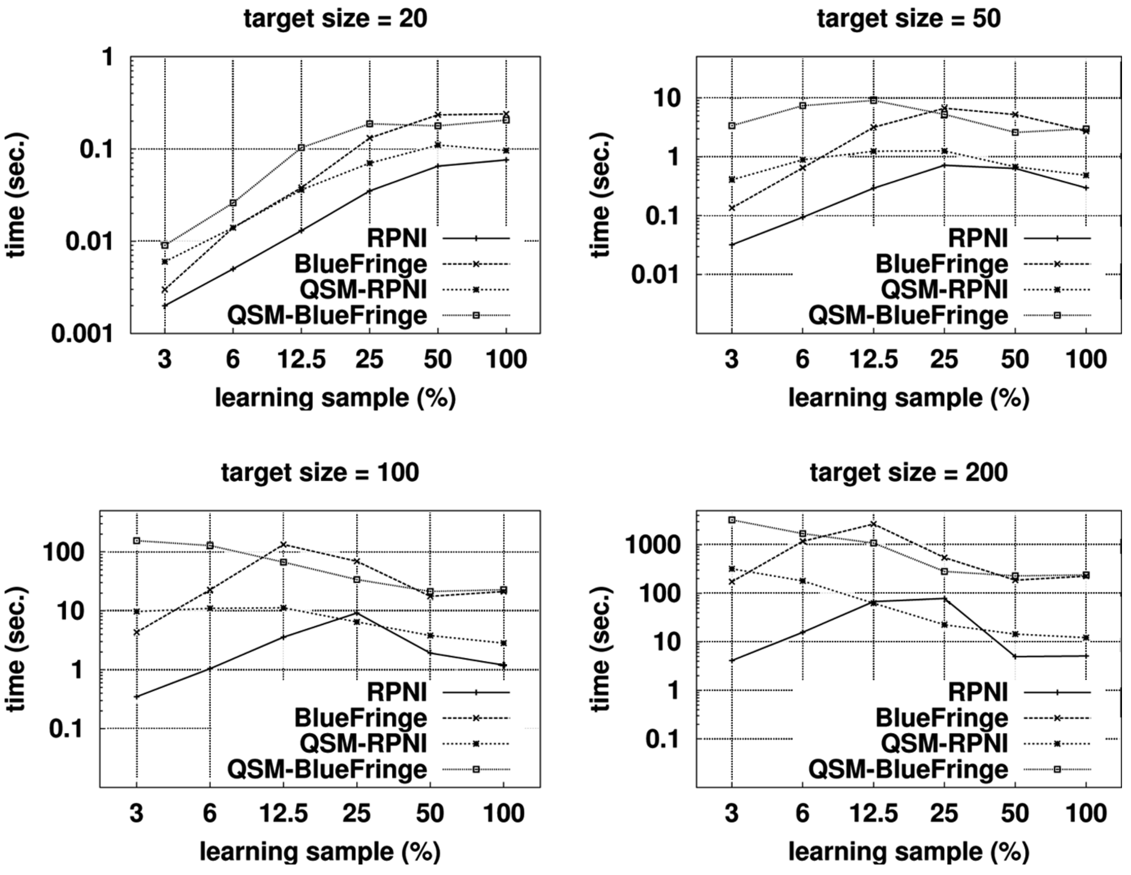

FIGURE 16 Induction time.

time follows the increase of the PTA size. When the learning sample becomes richer, better generalizations can be obtained by merging states in a more sound way. The classification rates of new data increases while the learning time reaches its maximum. The last phase is observed when the algorithm rapidly converges to a good model. Classification accuracy tends to 100% and the CPU time is decreased because the right merging operations are performed directly.

These known tendencies of the RPNI and Blue Fringe algorithms are confirmed in our experiments. They also apply to the QSM algorithm but left shifted with respect to the learning sample size. As already observed in Figure 14, convergence is indeed faster for the QSM algorithm. The relative time performance of the QSM algorithm with respect to RPNI or Blue Fringe depends on two contradictory effects. On the one hand, whenever a string is classified as negative by the oracle, QSM is called recursively on an extended sample. Each new call increases the CPU time as it could be considered as a new run of the RPNI or Blue Fringe algorithm. This run can be interrupted, however, and replaced by another one if a new negative example is included after an additional query. On the other hand, due to its faster convergence, QSM can obtain better results with fewer data originally provided.

The CPU times should thus be compared while considering the relative classification results of the various approaches. For instance, when the target size is 200 and 3% of the full training sample is used, QSM-Blue Fringe runs an order of magnitude slower than Blue Fringe. However, the classification accuracy of QSM-Blue Fringe is 95%, while it is 67% for Blue Fringe for the same amount of data. When the training size increases QSM-Blue Fringe actually becomes slightly faster than Blue Fringe because it has already nearly converged to the optimal solution.

DISCUSSION AND RELATED WORKS

The approach described in this article induces a behavior model of a software system from positive and negative scenarios and membership queries submitted to an end user. Additional domain knowledge is useful to reduce the number of queries but this knowledge is optional.

Alternative approaches to the synthesis of such models from scenarios have been proposed in the RE literature, but all of them require additional input information: a high-level message sequence chart showing how low-level MSC scenarios are to be flow-charted (Uchitel, Kramer, and Magee 2003), pre- and post-conditions of interactions expressed on global state variables (van Lamsweerde and Willemet 1998; Whittle and Schumann 2000), local MSC conditions (Kruger et al. 1998) This additional information is difficult to formulate for an end user.

The approach described in Mäkinen and Systä (2001) is, to the best of our knowledge, the unique previous attempt to use grammatical inference techniques for modeling software system behaviors. It is inspired from the model of a minimally adequate teacher proposed in Angluin (1987). We see two important limitations of this approach. Firstly, the inputs are not only global system scenarios but also state machine traces local to some specific agents. These traces are the sequences of events seen by a single agent (e.g., the events starting from or leading to a single vertical line in Figure 1). The alphabet of possible events and the learning strings are then specific to each agent and the learning problem has to be tackled separately for each agent while losing a global view of the system. Practical tests with our approach have shown that the learning task becomes simpler if one restricts it to the induction of separate agent models. This is much less adequate, however, from an RE perspective since it requires the end user to be able to classify traces in a different target language for each agent. It is much more convenient to interact with the end user in the scenario language of the global system because this is the language she used in the first place. Secondly, the end user is asked to answer both membership and equivalence queries. As claimed by Angluin herself, the availability of such a well-informed oracle is likely to be problematic in practice. She proposed to replace equivalence queries by a random sampling oracle in the PAC model (Valiant 1984). However, the sampling process is likely to require a large collection of scenarios and, in any case, this alternative was not followed by Mäkinen.

The possibility of learning with membership queries only has been proposed in Angluin (1981) with some theoretical limits but also positive results. In particular, the number of queries is shown to be exponential in $|Q|$, the size of the target model, if the only known information is indeed $|Q|$. The proof of such a result is based on a worst-case analysis which need not be relevant on average in practice. Moreover, the number of required membership queries is shown to be polynomial in $|Q|$ if the oracle receives a representative sample of the target language L. A structurally complete sample for the canonical automaton $A(L)$ (see Definition 3) is a representative sample of L.

There is no theoretical guarantee that the initial collection of positive scenarios provided by an end user in our approach forms a structurally complete sample with respect to the target model. However, the lack of structural completeness can be automatically detected in some useful cases in our application domain. In a global LTS, such as the one represented in Figure 13, there must be a continuation from any state (no deadlock). This property can be checked in the induced model and additional queries can be generated to the end user to ask for possible continuations from any state that would fail to satisfy this property. Finally, the practical experiments reported in Section 6.1 illustrate that the number of queries actually submitted to an end user for these test cases is always limited with the blue-fringe strategy.

The QSM algorithm described in Section 4 is called recursively on an extended scenario collection whenever a generated scenario query is classified as negative by the end user. An alternative approach would implement the incremental version described in Dupont (1996) to update the search space and the current solution whenever new positive or negative examples are added. This refinement has not been implemented in the ISIS tool because, even without it, a real-time execution is already guaranteed on representative test cases. This incremental version also relies on the original RPNI search order and it might be a bit challenging to extend it to the blue-fringe strategy.

The idea of incorporating domain knowledge by defining equivalence classes on the states of the initial machine and by propagating these constraints during the learning process has already been proposed in Coste et al. (2004). The novelty in our approach is the specific kinds of domain knowledge considered here (fluents, goals and domain properties, external system models) and the way they are reformulated as equivalence classes.

The technique described in Coste and Nicolas (1998) back-propagates state labels, initially only defined for positive or negative accepting states, and maintains these constraints while reducing the search for a DFA. The implementation in our ISIS tool goes a step further by back-propagating all constraints on the fly whenever a merging operation is observed compatible. This strategy implies to propagate more constraints at run-time but reduces the cost of a full initial propagation which would prevent merging attempts that might never be actually considered.

A first version of our approach is described in Damas et al. (2005, 2006) but the focus was specific to the software engineering community while the present article is formulated with a much more detailed learning perspective. The experimental evaluations (Section 6) have been significantly extended. Section 6.2 studying how the QSM algorithm scales with the size of the target machine, the discussion in Section 1.1, and the present section are all new. The revised pseudo-code of the QSM algorithm given in Algorithm 1 also better matches the actual implementation in the ISIS tool.

CONCLUSION AND PERSPECTIVES

This article presents an application of grammatical inference techniques to learn software requirements from end-user scenarios. Elicitation of software requirements is addressed as a DFA induction problem from positive and negative strings. These strings consist of interactions between agents of the software-to-be and of its environment. The end user is also assumed to be an oracle who can answer membership queries.

We propose here the QSM algorithm that extends the RPNI and blue-fringe algorithms with membership queries. This state-merging approach can be further constrained by incorporating domain knowledge to limit the search for the target machine while reducing the number of queries. This prior knowledge is formalized here as fluents, goals, and domain properties expressed in fluent linear temporal logic, and state models of external software components. Whenever available, domain knowledge offers the additional advantage of guaranteeing that the induced model is consistent with it.

The time complexity of QSM is a polynomial function of the learning sample size. Theoretical limits and practical issues of learning with queries are discussed. The QSM algorithm is implemented in the ISIS tool. We report practical evaluations on standard test cases in requirements engineering. Additional experiments on synthetic data show that the QSM algorithm scales well when the size of the target machine increases. In particular, the QSM algorithm systematically offers better generalization accuracy as compared to the RPNI and blue-fringe algorithms for sparse learning data.

Our future work includes the consideration of scenarios that would not always start from the initial state. From the learning perspective this case corresponds to learning a language from partial strings. The domain properties considered in this work are likely to be useful to suggest which state each partial scenario should be considered to start from.

Other application domains could also benefit from the particular formalization of domain knowledge proposed here and its translation into a set of constraints on the induction process.

An important open question is robustness to possible misclassifications by the end user. Traditional ways to deal with noisy inputs include probabilistic learning methods. The availability of domain knowledge and model checking techniques could also help to correct such mistakes.

REFERENCES

Angluin, D. 1978. On the complexity of minimum inference of regular sets. *Information and Control* 39:337–350.

Angluin, D. 1981. A note on the number of queries needed to identify regular languages. *Information and Control* 51:76–87.

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

Bell, T. and T. Thayer. 1976. Software requirements: Are they really a problem? In: *2nd International Conference on Software Enginering (ICSE)*, pp. 61–68. San Francisco, CA.

Boehm, B. 1981. *Software Engineering Economics.* Upper Saddle River, NJ: Prentice-Hall.

Brooks, F. 1987. No silver bullet: Essence and accidents of software engineering. *IEEE Computer* 20(4):10–19.

Coste, F., D. Fredouille, C. Kermorvant, and C. de la Higuera. 2004. Introducing domain and typing bias in automata inference. *Grammatical Inference: Algorithms and Applications*, pp. 115–126. Athens, Greece: Springer-Verlag.

Coste, F. and J. Nicolas. 1998. How considering incompatible state mergings may reduce the DFA induction search tree. *Grammatical Inference, ICGI'98*, pp. 199–210. Ames, IO: Springer-Verlag.

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

Damas, C., B. Lambeau, and A. van Lamsweerde. 2006. Scenarios, goals, and state machines: A win–win partnership for model synthesis. *International ACM Symposium on the Foundations of Software Engineering*, pp. 197–207. Portland, OR.

Dupont, P. 1996. Incremental regular inference. In: *Grammatical Inference: Learning Syntax from Sentences, ICGI'96*, pp. 222–237. New York: Springer-Verlag.

Dupont, P., L. Miclet, and E. Vidal. 1994. What is the search space of the regular inference? *Grammatical Inference and Applications, ICGI'94*, pp. 25–37. Alicante, Spain: Springer-Verlag.

Giannakopoulou, D. and J. Magee. 2003. Fluent model checking for event-based systems. In: *9th European Software Engineering Conference/11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 257–266. Helsinki.

Gold, E. 1978. Complexity of automaton identification from given data. *Information and Control* 37: 302–320.

Hall, R. and A. Zisman. 2004. Omml: A behavioral model interchange format. In: *12th IEEE Joint International Requirements Engineering Conference*, pp. 272–282, Kyoto, Japan.

Hopcroft, J. and J. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Reading, MA: Addison-Wesley.

Joseph, M. 1996. *Real-Time Systems: Specification, Verification and Analysis*. Upper Saddle River, NJ: Prentice-Hall.

Kruger, I., R. Grosu, P. Scholtz, and M. Broy. 1998. From mscs to statecharts. *International Workshop on Distributed and Parallel Emebedded Systems*, pp. 61–71, Scholoß Eringerfeld, Germany: Kluwer.

Lang, K. 1992. Random DFA's can be approximately learned from sparse uniform examples. In: *5th ACM Workshop on Computational Learning Theory*, pp. 45–52, Pittsburgh, PA, USA.

Lang, K., B. Pearlmutter, and R. Price. 1998. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: *Grammatical Inference*, pp. 1–12. Ames, IO: Springer-Verlag.

Magee, J. and J. Kramer. 1999. *Concurrency: State Models and Java Programs*. New York: Wiley.

Mäkinen, E. and T. Systä. 2001. Mas – an interactive synthetiser to support behavorial modeling in UML. In: *27th International Conference on Software Engineering*, pp. 15–24, Toronto, Canada.

Oncina, J. and P. García. 1992. Inferring regular languages in polynomial update time. In: *Pattern Recognition and Image Analysis*, eds. N. Pérez de la Blanca, A. Sanfeliu, and E. Vidal, Vol. 1, Series in Machine Perception and Artificial Intelligence, pp. 49–61. Singapore: World Scientific.

Oncina, J., P. García, and E. Vidal. 1993. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15(5):448–458.

The Standish Group. 1995. Software chaos. http://www.standishgroup.com. Last accessed 21 January 2008.

Uchitel, S., J. Kramer, and J. Magee. 2003. Synthesis of behavorial models from scenarios. *IEEE Transactions on Software Engineering* 29(2):99–115.

Valiant, L. 1984. A theory of the learnable. *Communications of the Association for Computing Machinery* 27(11):1134–1142.

van Lamsweerde, A. 2001. Goal-oriented requirements engineering: A guided tour. In: *5th Intl. Symp. Requirements Engineering (RE)*, pp. 249–263. Toronto: IEEE Press.

van Lamsweerde, A. 2004. Goal-oriented requirements engineering: A roundtrip from research to practice. In: *12th IEEE Joint International Requirements Engineering Conference*, pp. 4–7, Kyoto, Japan.

van Lamsweerde, A. and L. Willemet. 1998. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering* 24(12):1089–1114.

Whittle, J. and J. Schumann. 2000. Generating statechart designs from scenarios. In: *22nd International Conference on Software Engineering*, pp. 314–323. Limerick, Ireland.

NOTES

1. A counterexample in this context is a string belonging either to the language of the proposed model or to the target language but not both languages.
2. Let n be the number of states of $PTA(S_+)$. By construction, $n \in \mathcal{O}(||S_+||)$. The search space size is the number of ways a set of n elements can be partitioned into nonempty subsets. This is called a Bell number *B(n)*. It can be defined by the Dobinski's formula: $B(n) = \frac{1}{e}\sum_{k=0}^{\infty} \frac{k^n}{n!}$. This function grows much faster than 2^n.
3. The standard order of strings on the alphabet $\Sigma = \{a, b\}$ is $\lambda < a < b < aa < ab < ba < bb < aaa < \cdots$
4. As explained in Section 2.1, any proper prefix of a negative scenario is a positive scenario. Hence, the initial PTA also stores all the proper prefixes of the negative scenarios. Besides, an augmented $PTA(S_+, S_-)$ with positive and negative strings is considered in Section 4.3.
5. The assignment in the corresponding **while** loop is assumed to be *true* whenever a valid state pairs is returned by `ChooseStatePairs`. When no more state pairs are considered for merging, `ChooseStatePairs` returns *(nil,nil)* and the assignment is evaluated to *false*. This abuse of notation in the pseudo-code allows to keep it more concise. A similar remark also applies to the inner **while** loop of the QSM algorithm.
6. A more sophisticated strategy to update the current solution in such a case is mentioned in Section 7.
7. All states in a LTS are (positively) accepting. Hence, all states of $PTA(S_+)$ are positive accepting states. Since a negative scenario is the concatenation of a positive prefix with a single additional symbol, the augmented $PTA(S_-, S_-)$ only contains states labeled either as positive accepting or negative accepting.
8. The fact that the number of fluents and goals is the same for each case study is purely coincidental.