Generating Annotated Behavior Models from End-User Scenarios

Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde, *Member*, *IEEE*

Abstract—Requirements-related scenarios capture typical examples of system behaviors through sequences of desired interactions between the software-to-be and its environment. Their concrete, narrative style of expression makes them very effective for eliciting software requirements and for validating behavior models. However, scenarios raise coverage problems as they only capture partial histories of interaction among system component instances. Moreover, they often leave the actual requirements implicit. Numerous efforts have therefore been made recently to synthesize requirements or behavior models inductively from scenarios. Two problems arise from those efforts. On the one hand, the scenarios must be complemented with additional input such as state assertions along episodes or flowcharts on such episodes. This makes such techniques difficult to use by the nonexpert end-users who provide the scenarios. On the other hand, the generated state machines may be hard to understand as their nodes generally convey no domain-specific properties. Their validation by analysts, complementary to model checking and animation by tools, may therefore be quite difficult. This paper describes tool-supported techniques that overcome those two problems. Our tool generates a labeled transition system (LTS) for each system component from simple forms of message sequence charts (MSC) taken as examples or counterexamples of desired behavior. No additional input is required. A global LTS for the entire system is synthesized first. This LTS covers all scenario examples and excludes all counterexamples. It is inductively generated through an interactive procedure that extends known learning techniques for grammar induction. The procedure is incremental on training examples. It interactively produces additional scenarios that the end-user has to classify as examples or counterexamples of desired behavior. The LTS synthesis procedure may thus also be used independently for requirements elicitation through scenario questions generated by the tool. The synthesized system LTS is then projected on local LTS for each system component. For model validation by analysts, the tool generates state invariants that decorate the nodes of the local LTS.

Index Terms—Scenario-based elicitation, synthesis of behavior models, scenario generation, invariant generation, labeled transition systems, message sequence charts, model validation, incremental learning, analysis tools.

1 Introduction

Scenarios are widely recognized as an effective means for requirements elicitation, documentation, and validation [11]. They support an informal, narrative, and concrete style of description that focuses on the dynamic aspects of software-environment interaction. Scenarios are therefore easily accessible to practioners and stakeholders involved in the requirements engineering process [30].

The context of this paper is a project aimed at automating the production of Web applications from end-user scenarios. In this project, behavioral models need to be obtained from scenarios as an intermediate product for generating code fragments that populate a predefined architecture for the application.

A *scenario* is a temporal sequence of interactions among system components. The word *system* refers here to the software-to-be together with its environment. A system is made of active components, called *agents*, that control system behaviors. Some agents form the environment, others form the system-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. A scenario *episode* is an interaction subsequence that achieves some objective (generally left implicit). *Positive scenarios* describe typical examples of desired interactions, whereas *negative scenarios* describe undesired ones.

This paper addresses the problem of synthesizing a state machine model of the system from scenarios submitted by end-users. For end-user involvement, we need to put strong requirements on the synthesis process.

- The input to the generation algorithm should be a set of end-users scenarios and *end-user scenarios only*. Such users are most likely to be unable to provide additional input such as state assertions along scenario episodes or flowcharts on such episodes. The scenarios should, moreover, be expressed in some simple, "box-and-arrow" form.
- Both positive and negative scenarios should be taken into account. Our experience in a variety of requirements engineering projects over the years showed that negative scenarios are quite common among the examples provided by stakeholders.
- Scenarios are inherently incomplete (like examples or test data are). The generation algorithm should support the elicitation of additional, "interesting" positive/negative scenarios that are not originally provided by the end-user.

- *The authors are with the Department of Computing Science and Engineering (INGI), Université Catholique de Louvain, Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium.*
E-mail: {damas, blambeau, pdupont, avl}@info.ucl.ac.be.

Manuscript received 6 May 2005; revised 3 Nov. 2005; accepted 7 Nov. 2005; published online 22 Dec. 2005.
Recommended for acceptance by S. Uchitel, M. Broy, I.H. Krüger, and J. Whittle.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSI-0141-0505.

- The synthesis of state machine models from scenarios should be incremental; the models should be incrementally refinable as further scenarios become available.
- In view of possible overgeneralizations and incomplete/inconsistent scenarios, the synthesized state machine models should be understandable for validation and correction by the analyst before code generation starts.
- The generated models should be well-structured for high-quality code generation, in particular, through one state machine per software agent.

Various efforts have been reported in the literature to generate behavior models from scenarios. We review them briefly with respect to the above requirements.

Uchitel et al. developed a technique for generating one labeled transition system (LTS) for each agent of a message sequence chart (MSC) specification [29]. Their approach requires additional input, namely, a high-level message sequence chart (hMSC) that specifies how the MSC scenarios are to be flowcharted. In our experience, such hMSC may become quite complex for nontoy systems. Adding a new MSC in the specification may require some nontrivial refactoring of the original hMSC [20]. Asking end-users to provide a correct and complete hMSC as input thus seems unrealistic. The LTS synthesis algorithm does not take negative scenarios into account. Moreover, the synthesized LTS are not easily understandable by humans as their states are labeled by numbers only.

Whittle and Schumann proposed a technique for generating UML statecharts from sequence diagrams that capture positive scenarios—and positive scenarios only [31]. Their technique requires scenario interactions to be annotated by pre and postconditions on global state variables expressed in the Object Constraint Language (OCL). In a similar spirit, Kruger et al. proposed a technique for translating MSCs into statecharts [15]. Their technique also requires state information as additional input (in this case, through MSC conditions). It is unclear in both approaches whether end-users are able to provide such additional information.

Mäkinen and Systä developed an interactive approach for synthesizing UML statecharts from sequence diagrams that capture positive scenarios [23]. Their so-called Minimally Adequate Synthesizer (MAS) uses grammatical inference and asks the user trace questions in order to avoid undesirable generalizations. (A trace question is a path in the state machine local to a specific agent.) MAS focuses on single agents; generalization must therefore be done independently for each software agent. Trace questions may be quite hard to understand by end-users as they do not show global system behaviors. Overgeneralization may frequently occur in view of the built-in assumption that trace events with the same label lead to the same component state (unless a counterexample is specified). To eliminate such poor generalization, the user has to understand and validate the agent's generated state machine and provide state machine traces as counterexamples to indicate undesired behaviors and restart the generalization process.

Van Lamsweerde and Willemet developed an inductive learning technique for generating goal specifications in linear temporal logic (LTL) from positive and negative scenarios expressed in MSC-like form [17]. Büchi automata could then be generated from the LTL specifications using known algorithms; the resulting state machines would, however, be very hard to understand for validation. Moreover, as in [31], the user has to provide pre/postconditions of scenario interactions.

Other efforts have been devoted to producing SDL specifications from MSCs (e.g., [9]). The techniques proposed there require complex forms of MSC as input to the synthesis process. Such MSCs cannot be considered as scenarios expressed at requirements engineering time by end-users. No negative information is exploited.

This paper presents techniques, supported by a tool, that meet the above requirements for state machine generation from end-user scenarios.

Our approach takes both positive and negative scenarios as input. We synthesize an LTS covering all positive scenarios and excluding all negative ones. The synthesis procedure extends grammar induction techniques developed in [26], [4]. Our inductive learning procedure is interactive and incremental on training instances, which makes it possible to integrate missing scenarios and scenario corrections on-the-fly. It requires no additional state or flowchart information. The original set of scenarios is incrementally completed by asking the user scenario questions that are generated during synthesis. A *scenario question* consists of showing the user a specific scenario and asking her to classify it as positive or negative. The synthesized LTS is then transformed into a parallel composition of finer LTS—one LTS per agent.

To enable validation and documentation of the resulting behavior model, state invariants are generated as node decorations from fluent definitions to be provided by the analyst. Fluents are state predicates whose truth values are determined by the occurrences of initiating and terminating events [5]; they provide a nice interface between goal specifications and goal operationalizations [16] and can easily be identified from goal formulations.

The paper is organized as follows: Section 2 presents some required background on scenario specifications, labeled transition systems, fluents, and grammar induction. Section 3 presents an overview of the various steps supported by our tool. Section 4 details the process of synthesizing LTS models and generating scenario questions. Section 5 details the fluent-based invariant generation procedure. The entire approach is illustrated in Section 6 on a nontrivial case study, a mine pump control system [13], [14].

2 Background

To make the paper self-contained, this section introduces some basic material on message sequence charts (MSC), labeled transition systems (LTS), fluents, and grammar induction. A simple train system fragment will be used throughout the paper as a running example to illustrate the various techniques. 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,

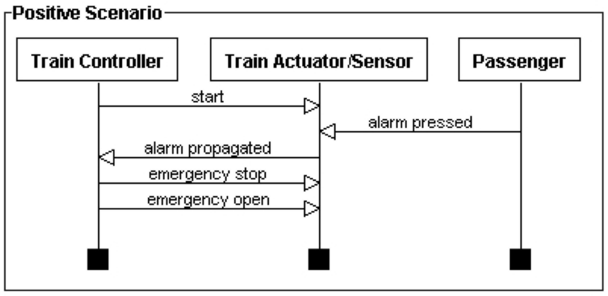

Fig. 1. Positive scenario for a train system.

the controller must stop the train first and then open the doors in an emergency.

2.1 Scenarios as Message Sequence Charts

A simple MSC language 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 the 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. The event is synchronously controlled by the source agent and monitored by the target agent.

An MSC timeline defines a total ordering on incoming/outgoing events, whereas an entire MSC defines a partial ordering on all events. To allow end-users to submit their scenarios, we limit the input language to be a very simple one, leaving aside more sophisticated MSC features such as conditions, timers, coregions, etc.

Fig. 1 shows a MSC capturing the following scenario: *"The train is started by the controller; the latter then waits for external stimuli. A passenger presses the alarm button; the alarm is propagated to the controller; the latter then stops the train and opens the doors in an emergency."*

Scenarios are positive or negative. A positive scenario illustrates some desired system behavior. A negative scenario captures a behavior that may not occur. It is captured 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.

Fig. 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 (red) dashed line in our tool. The scenario in Fig. 2 is used to express that the train controller may not open the doors after having started the train (without any intermediate interaction).

The semantics of MSCs used in this paper is the one introduced in [29]. As this semantics is defined in terms of labeled transition systems and parallel composition, we come back to it in Section 2.2, where LTS are introduced.

2.2 State Machines as Labeled Transition Systems

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

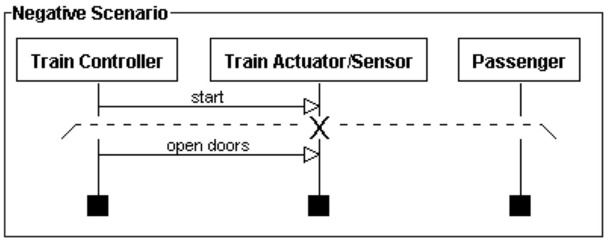

Fig. 2. Negative scenario for a train system.

between states. Each transition is labeled by an event. The state machines in this paper are a particular class of automata called labeled transitions systems [21].

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 deterministic if, for any q in Q and any e in Σ, $\delta(q, e)$ has at most one member.

In a *labeled transitions system* (LTS), all states are accepting states, that is, the sets Q and F are the same. An LTS is therefore simply denoted by a 4-tuple (Q, Σ, δ, q_0). The alphabet Σ corresponds to the set of event labels of the LTS. In an LTS, if a state q has no outgoing transition with label l, no event with label l can occur when the system is in state q.

A *finite execution* of an LTS (Q, Σ, δ, q_0) is a finite sequence of events $\langle e_1, \ldots, e_n \rangle$, with $e_i \in \Sigma$, accepted by the LTS from its initial state. Such an execution is said to finish in state q if the LTS is in state q after having performed that event sequence from the initial state. Prefixes of a finite execution are finite executions as all LTS states are accepting states.

Complex systems can be modeled through parallel composition of LTS components [24]. The parallel composition of two LTS P and Q, denoted by $P \| Q$, models their joint behavior. The composed model behaves asynchronously, but synchronizes on shared events. A system composed of agents $a_1, \ldots, a_n$ modeled by LTS $A_1, \ldots, A_n$ is thus modeled by the LTS $A_1 \| \ldots \| A_n$.

The semantics of MSCs can be defined in terms of LTS and parallel composition [29]. An MSC timeline defines a total order on its input and output events. Therefore, it defines a unique finite LTS execution that captures a corresponding agent behavior. Fig. 3 shows the LTS behavior corresponding to the Train Controller agent in Fig. 1 The semantics of an entire MSC can similarly be defined in terms of the LTS modeling the entire system. As MSCs define partial orders of their events, we need to consider MSC linearizations of such partial orders [1]. A linearization defines a total order of events and represents one temporal behavior of the system. In the context of end-users scenarios, we consider finite MSCs, that is, MSCs with finite sets of linearizations. An MSC linearization defines a finite execution of the system's LTS. An entire MSC then defines a finite set of such executions. Positive MSCs define desirable executions, whereas negative MSCs define

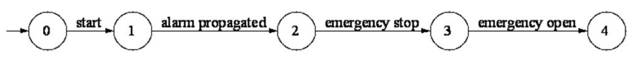
$$\begin{tikzcd} \arrow[r] & 0 \arrow[r, "start"] & 1 \arrow[r, "alarm\ propagated"] & 2 \arrow[r, "emergency\ stop"] & 3 \arrow[r, "emergency\ open"] & 4 \end{tikzcd}$$

Fig. 3. Finite LTS execution for the Train Controller agent in the *Positive Scenario* in Fig. 1.

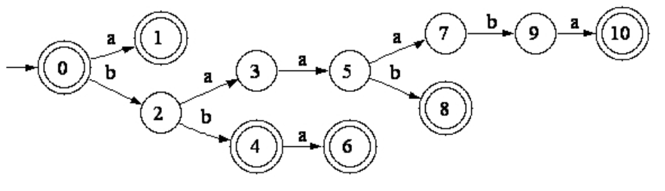

Fig. 4. The prefix tree acceptor build from $S+ = \{\lambda, a, bb, bba, baab, baaaba\}$.

rejected ones. (To simplify the presentation, the MSC examples in this paper have one linearization only.)

2.3 Interfacing Event-Based and State-Based Models through Fluents

Miller and Shanahan define fluents as "*time-varying properties of the world that are true at particular time-points if they have been initiated by an event occurrence at some earlier time-point, and not terminated by another event occurrence in the meantime. Similarly, a fluent is false at a particular time-point if it has been previously terminated and not initiated in the meantime*" [25].

A *fluent Fl* is a proposition defined by a set $Init_{Fl}$ of initiating events, a set $Term_{Fl}$ of terminating events, and an initial value $Initially_{Fl}$ that can be true or false. The sets of initiating and terminating events must be disjoint. The concrete syntax for fluent definition is the following [5]:

$$\text{fluent } Fl =< InitFl, TermFl > \text{ initially } Initially_{Fl}.$$

In our train example, the safety goal "*Doors shall remain closed while the train is moving*" suggests two fluents: *moving* and *doorsClosed*. The former is defined as follows:

$$\text{fluent } moving =< \{start\}, \{stop, emergency\ stop\} > \text{ initially } false.$$

2.4 Grammar Induction

Inductive learning aims at finding a theory that generalizes a set of observed examples. In *grammar induction*, the theory to be learned is a formal language and the set of positive examples is made of strings defined on a specific alphabet. A negative sample corresponds to a set of strings not belonging to the target language. When the target language is regular and the learned language is represented by a deterministic finite state automaton (DFA), the problem is known as DFA induction.

2.4.1 DFA Identification in the Limit

Identification in the limit is a learning framework in which an increasing sequence of strings is presented to the learning algorithm [6]. The strings are randomly drawn and correctly labeled as positive or negative. Learning is successful if the algorithm infers the target language in finite time after having seen finite samples. This framework justifies why successful DFA learning needs both positive and negative strings. Gold showed that the class of regular languages cannot be identified in the limit from positive strings only [6]. In practice, convergence in finite time toward an exact solution is often bargained with reasonably fast convergence toward a good approximate solution [18].

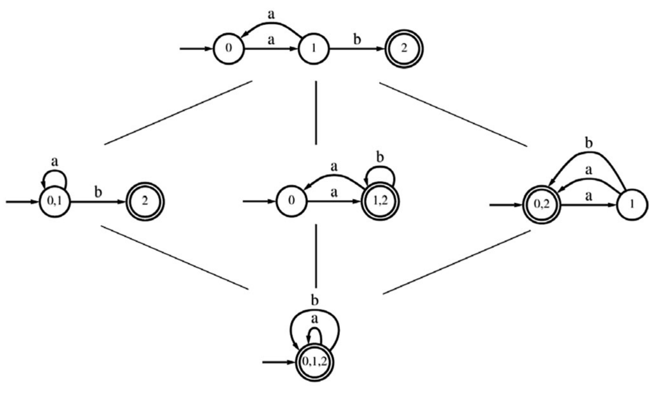

Fig. 5. A lattice of partitions defining quotient automata.

2.4.2 The Search Space of DFA Induction

DFA induction requires efficient search through the space of possible generalizations. To characterize the search space, we need to recall some links between finite automata and regular languages.

Let Σ denote a finite alphabet, u, v, w denote strings over Σ, and let λ denote the empty string. A string u is a prefix of v if there exists a string w such that $uw = v$. A language L is any subset of the set Σ^* of strings over Σ.

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. For any regular language L, the canonical automaton $A(L)$ is the minimal DFA accepting L, that is, the DFA having the smallest number of states and accepting L. The automaton $A(L)$ is known to be unique up to state renumbering [7].

A positive sample $S+$ can be represented by a *prefix tree acceptor* $PTA(S+)$ as depicted in Fig. 4 (accepting states there are represented by double circles); $PTA(S+)$ is the largest DFA accepting $S+$ exactly. Learning an automaton A by generalizing a positive sample can be performed by merging states from $PTA(S+)$. Such generalization is defined through a *quotient automaton*, constructed by partitioning the states of A.

Consider, for example, the automaton A represented at the top of Fig. 5. Let $\pi = \{\{0, 2\}, \{1\}\}$ be a partition defined on its state set $Q = \{0, 1, 2\}$. Its quotient automaton with respect to the partition π, denoted A/π, is represented on the right. Any accepting path in A is also an accepting path in its quotient automaton. In other words, merging states in an automaton generalizes the language it accepts. As a quotient automaton corresponds to a particular partition, the set of possible generalizations which can be obtained by merging states of an automaton A is defined by a lattice of partitions. Fig. 5 presents all quotient automata that can be derived from the automaton at the top.

Learning a language L aims at generalizing a positive sample $S+$ under the control of a negative sample $S-$, with $S+ \subseteq L$ and $S- \subseteq \Sigma^*$. This is made 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 2.1 (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 staring.*

Rather than a requirement on the sample, structural completeness should be considered as a limit on the possible generalizations that are 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. The following theorem is proven in [3] to characterize the search space of the DFA induction problem.

Theorem (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 $PTA(S+)/\pi = A(L)$.*

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. Finding the minimal DFA is an NP-complete problem [7].

2.4.3 The RPNI Algorithm and Its Convergence

The RPNI algorithm explores a very small fraction of the entire search space with the guarantee of finding the correct DFA when the learning sample is rich enough [26]. The convergence of RPNI on the correct automaton $A(L)$ is guaranteed when the algorithm receives a sample as input that includes a characteristic sample of the target language [4]. A proof of convergence is presented in [27] in the more general case of transducer learning. Some further notions are needed here.

Definition 2.2 (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, 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 automaton of Fig. 6 is $Sp(L) = \{\lambda, b\}$.

Definition 2.3 (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 automaton of Fig. 6 is $N(L) = \{\lambda, a, b, ba, bb\}$.

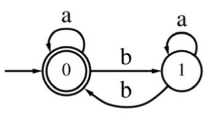

Fig. 6. An automaton A with $SP(L) = \{\lambda, b\}$ and $N(L) = \{\lambda, a, b, ba, bb\}$. The sample $S = (S+, S-)$ with $S+ = \{\lambda, a, bb, bba, baab, baaaba\}$ and $S = \{b, ab, aba\}$ is characteristic for A.

Definition 2.4 (Characteristic sample). *A sample $S = (S+, S-)$ is characteristic for language L and the RPNI algorithm if it satisfies the following conditions:*

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

Condition 1 guarantees that each element of the kernel belongs to $S+$ if it also belongs to the language or, otherwise, is the prefix of a string of $S+$. This condition can be seen to imply the structural completeness of the sample $S+$ with respect to $A(L)$. In this case, the DFA search space theorem guarantees that the automaton $A(L)$ belongs to the lattice derived from $PTA(S+)$. 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 $PTA(S+)$ with another state corresponding to an element y of the kernel is made incompatible by the existence of xu in $S+$ and yu in $S-$ or the converse.

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 Fig. 6.

3 Overview of the Approach

Fig. 7 shows the various steps of our approach as seen by users. We outline them first before providing the technical details in the next sections. In the first step, the end-user introduces positive and negative scenarios. In the second step, the tool synthesizes a LTS for the global system which covers all positive scenarios and excludes all negative ones. The generalization process is guided by scenario questions asked to the end-user and generated during the incremental synthesis process. The synthesized LTS is then projected to obtain each local agent LTS. In the third step, fluent definitions are provided by the analyst as optional input for generating state invariants to document and validate the generated LTS.

(Step 1) Submitting an initial set of positive and negative scenarios. The end-user has to provide a nonempty set of scenarios (positive and/or negative) as initial input to the process. All scenarios must start in the same initial state. Fig. 8 presents typical end-user scenarios for the train example. The initial scenario collection there contains three positive scenarios and one negative.

(Step 2) Generating scenario questions and synthesizing agent LTS. The tool incrementally generates and refines

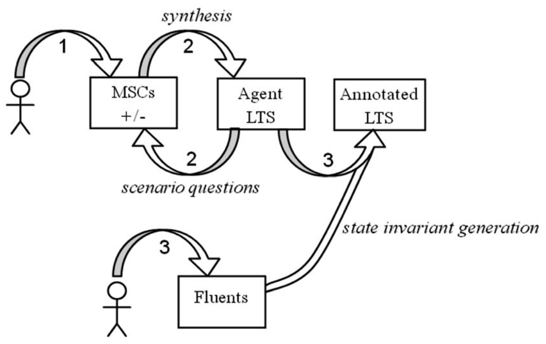

Fig. 7. Generating annotated state machines from end-user scenarios.

a global LTS for the system that covers positive scenarios and excludes negative ones. The generalization process is guided by scenarios generated during synthesis as questions to the end-user. The user just needs to classify those generated scenarios as being positive or negative. For the initial scenarios in Fig. 8, the tool generates three scenario questions while producing a first LTS sketch. These questions are shown in Fig. 9. Scenario questions are composed of a prefix and a suffix. The prefix is an already admissible behavior. The suffix must be accepted or rejected by the end-user. The prefix and suffix of a question are separated by a dashed line labelled with a question mark. The first scenario asks the user if the train controller can start after having started and stopped the train. The user should accept this scenario. The second question can be rephrased as follows: "*if the train starts and a passenger presses the alarm button, can the controller then open the doors in emergency and close the doors afterward?*" This scenario should be rejected as the train should not move with open doors. The third question asks the user if the passenger can press the alarm button in the initial state. The end-user might accept this scenario. Once they are accepted or rejected, the generated scenarios are added to the scenario collection as positive or negative ones, respectively.

Finally, the synthesized LTS is projected on each agent to obtain its LTS. Fig. 10a shows the generated Train Controller LTS at the end of this process.

(Step 3) Generating state invariants to document the generated state machines. For validation and documentation purposes, each state of the generated LTS may be decorated with a state invariant that holds at any time this state is visited. If this option is taken, fluent definitions are to be provided by the user from goal formulations. The user here is no longer the end-user, but the analyst who wants the state machines to be made comprehensible for documentation and validation before code generation. For the train controller, three fluents might be identified from goal formulations:

- fluent *moving* = <{*start*}, {*stop, emergency stop*} > initially *false*,
- fluent *doors_open* = <{*open doors, emergency open*}, {*close doors*}> initially *false*,
- fluent *alarmed* = <{*alarm propagated*}, {*emergency open*}> initially *false*.

The decorated state machine is shown in Fig. 10b. If the analyst finds problems with the generated state machines, she can do the following:

- If the state machine is overgeneralized, she should provide a negative scenario and restart the LTS synthesis process.
- If the state machine is incomplete, she may 1) change the state machine by hand (e.g., by adding a transition) or 2) add new positive scenarios and then restart the LTS synthesis process. Alternative 1) can result in inconsistencies between the LTS and the end-user's scenario collection; the analyst should therefore raise the problem to the end-user and modify the latter accordingly.

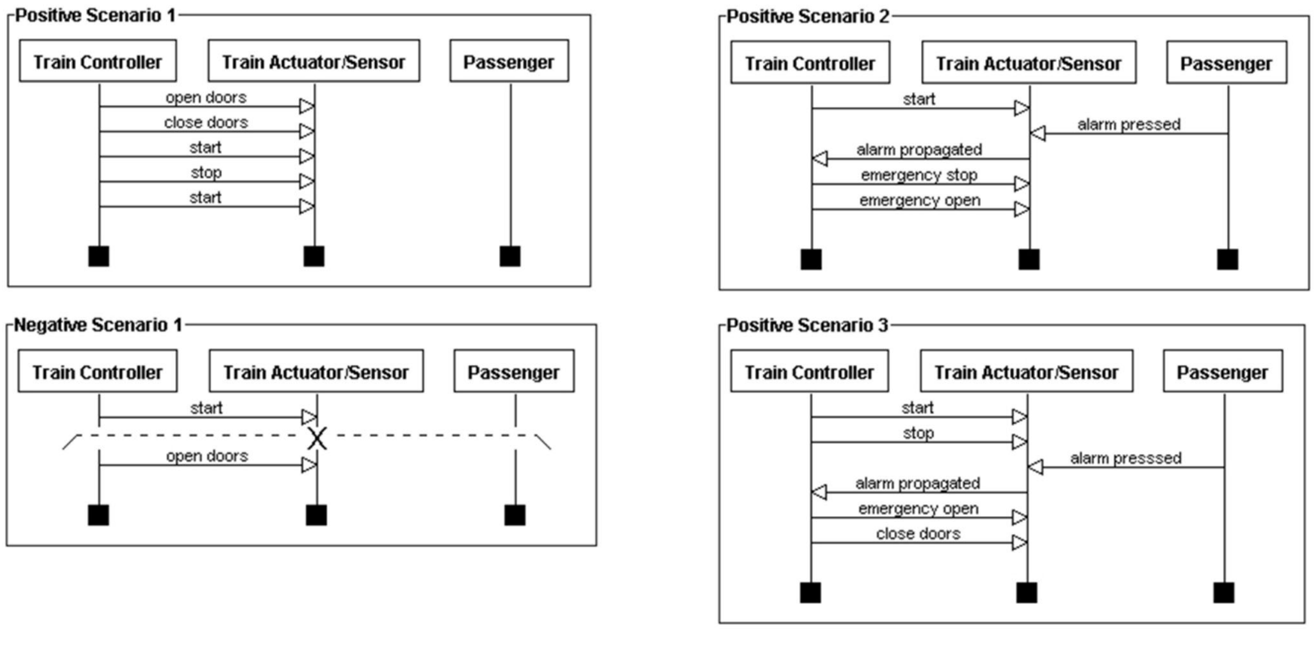

Fig. 8. End-user's positive and negative scenarios for a train system.

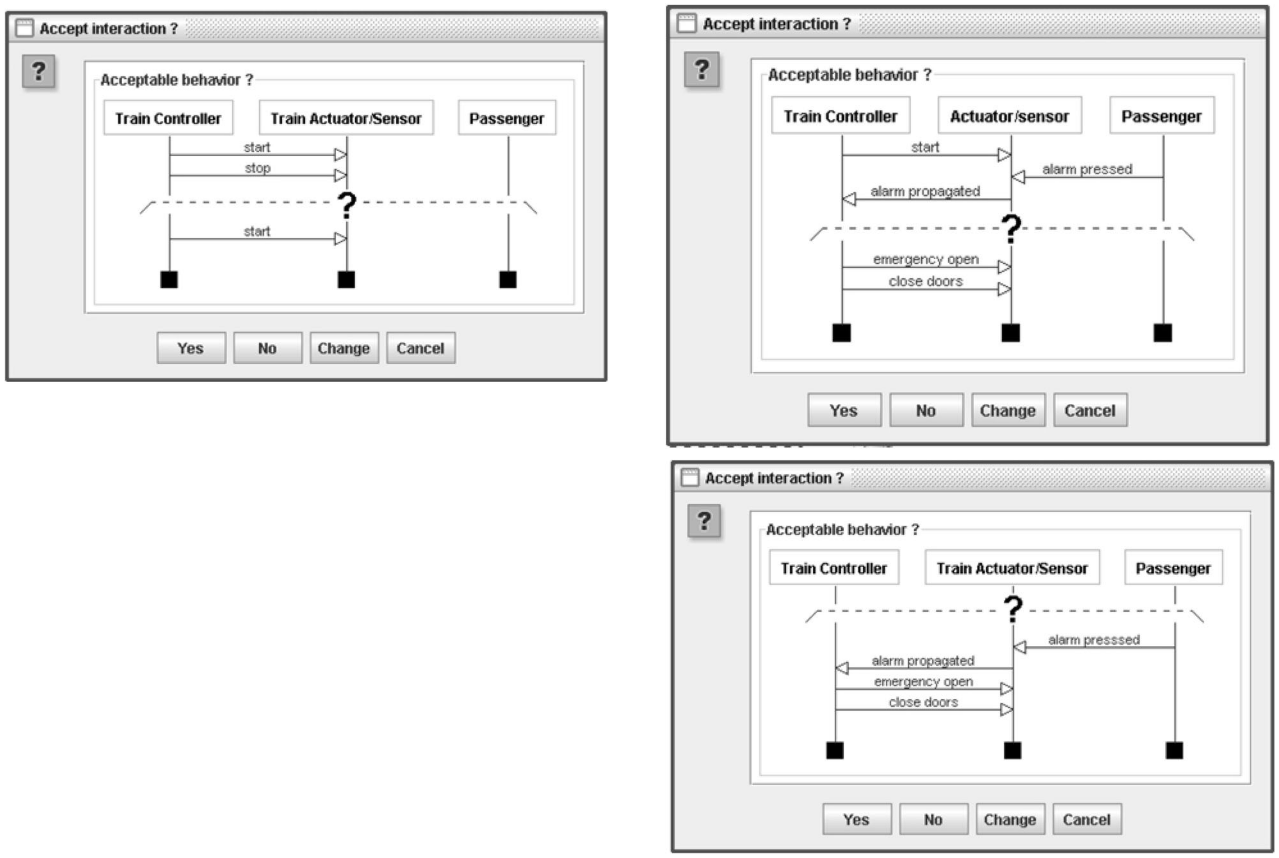

Fig. 9. Scenario questions generated to the end-user during the LTS synthesis process.

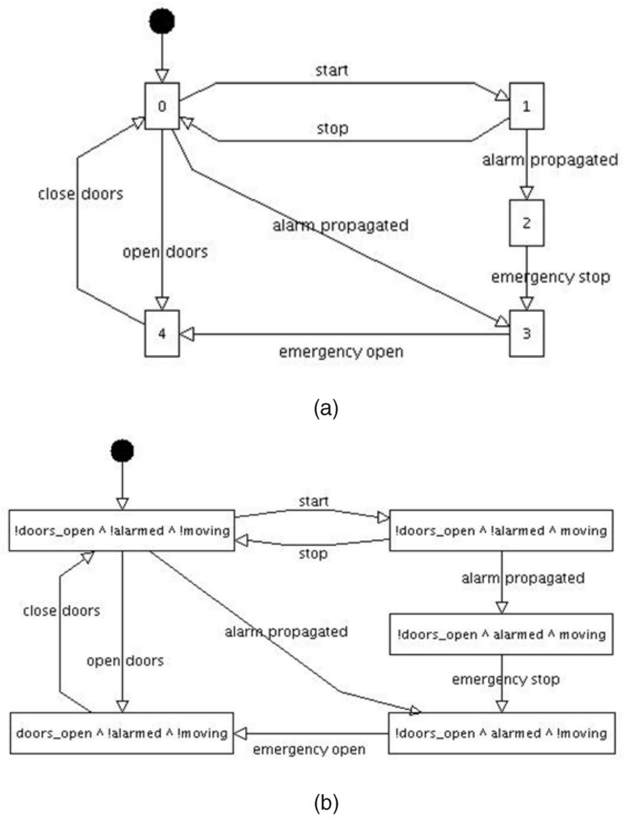

Fig. 10. Generated LTS of the Train Controller: (a) before decoration and (b) decorated with state invariants.

4 Synthesizing Labeled Transition Systems from End-User Scenarios

This section describes our technique for synthesizing labeled transition systems (LTS) from simple message sequence charts (MSC) without any extra information such as hMSCs or state assertions. Our algorithm extends an efficient automaton induction algorithm known as RPNI [26] to make it interactive through scenario questions. Section 4.1 explains how LTS synthesis can be achieved through automaton induction and why such interaction is needed in our context. The synthesis of a global system LTS is detailed in Section 4.2. The projection of this LTS on the agents forming the system is explained in Section 4.3. The properties of our approach are then discussed in Section 4.4.

4.1 LTS Synthesis as a Grammar Induction Problem

As introduced in Section 2.4, a regular language can be learned through automaton induction techniques. LTS are a particular class of automata that contain only accepting states. A positive (respectively, negative) MSC timeline defines a unique execution of the LTS associated with the corresponding agent. It therefore defines an accepted (respectively, rejected) string of the regular language represented by the agent LTS. In a similar way, a positive (respectively, negative) MSC linearization defines an accepted (respectively, rejected) string of the regular language generated by the LTS of the global system. The latter language will be denoted by $L(S)$ in the sequel.

Any prefix of a finite LTS execution is a valid LTS execution as LTS contain only accepting states. Therefore,

```
Input: A non-empty initial scenario collection $Sc = (S+, S-)$
Output: An automaton $A$ consistent with an extended
        collection $Sc = (S+, S-)$
$A \leftarrow$ Initialize($S_+$)
while $(q, q') \leftarrow$ ChooseStatePairs($A$) do
    $A_{new} \leftarrow$ Merge($A, q, q'$)
    if Compatible($A_{new}, S_-$) then
        ok ← true
        while $Q \leftarrow$ GenerateQuestion($A, A_{new}$) do
            if CheckWithEndUser($Q$) then
                $S_+ \leftarrow S_+ \cup Q$
            else
                $S_- \leftarrow S_- \cup Q$
                ok ← false
                break
        if ok then
            $A \leftarrow A_{new}$
return $A$
```

Fig. 11. An interactive adaptation of the RPNI induction algorithm.

any prefix of a positive MSC linearization is an accepted string of $L(S)$. Such a linearization thus provides a finite set of positive strings of $L(S)$. In a similar way, as the prefix of a negative MSC is a positive MSC, a negative MSC linearization provides one rejected string and a finite set of accepted strings of $L(S)$.

Our choice of an RPNI-based technique to overcome the absence of additional state information about the submitted scenarios is motivated by the following observation: If the end-user scenario collection contains a characteristic sample according to $L(S)$, an RPNI-based algorithm will ensure that $L(S)$ can be learned in polynomial time. The learned system LTS can then be projected on each agent, using standard automaton algorithms, in order to obtain their respective LTS.

In practice, the initial scenario collection might not provide a characteristic sample for the considered system (see the importance of negative strings in Definition 2.4). For example, an initial scenario collection with no negative example cannot be characteristic for any nontrivial system —that is, any system for which the target automaton contains at least two states (Condition 2 in Definition 2.4 states that the merging of those states should be made incompatible by at least one negative scenario). To overcome the problem of poor generalization when dealing with such limited training sets, we extend the RPNI algorithm so that it generates additional scenarios and asks the end-user to classify them as positive or negative. The learned system will cover all positive scenarios and reject all negative ones, including the interactively generated/classified ones.

4.2 Interactive Synthesis of the System LTS

Our interactive RPNI-based synthesis algorithm is given in Fig. 11. The algorithm takes a scenario collection as input and produces an LTS of the global system as output. The completion of the initial scenario collection with classified scenarios that were generated during synthesis is another output of the algorithm. The input collection must contain at least one positive or one negative scenario. The generated LTS 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 LTS covering all positives scenarios only, i.e., the latter does not introduce any generalization of the system behaviors described by the positive scenarios. The system 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 LTS, i.e., by successively computing quotient LTS from the initial one (see Section 2.4). The induction process is such that, at any step, the current quotient LTS covers all positive scenarios and excludes all negative ones, including the interactively classified ones. In the sequel, an LTS will be 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 will be said *compatible for merging* (respectively, *incompatible*) if the quotient LTS which results from their merging is compatible (respectively, *incompatible*) with the current set of scenarios.

Following the algorithm given in Fig. 11, the *Initialize* function returns an initial candidate LTS solution built from $S+$. Next, pairs of states are iteratively chosen from the current solution. 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 *GenerateQuestion* function and submitted to the end-user for classification (*CheckWithEndUser*). Scenarios classified as positive are added to the initial collection. When a generated scenario is classified as negative, it is added as negative example; the generation of the current solution is ended and the candidate quotient automaton is discarded. Otherwise, when all generated scenarios are classified as positive, the quotient automaton becomes the current candidate solution. The process is iterated until no more pairs of states can be considered for merging. The learned LTS is then returned as output of the algorithm.

This interactive algorithm has a polynomial time complexity in the size of the learning sample. Whenever a quotient automaton is considered compatible and the end-user classifies all generated scenarios as positive examples, the states that were merged remain merged forever. In other words, there is *no backtracking* in the induction process. This is a key feature explaining the time complexity of the algorithm.

We now have a closer look at the algorithm by detailing its various functions.

(*Initialize*) The *Initialize* function returns the initial solution built from $S+$ as prefix tree acceptor $PTA(S+)$ constructed from positive MSCs. The PTA built from the positive sample in Fig. 8 is shown on top of Fig. 12. According to the modeling hypothesis discussed before, all PTA states are accepting states. As mentioned before, we assume here that all scenarios are starting in the same system state.

(*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 lexicographical order "<" on strings. Each $PTA(S+)$ state can be labeled by its unique prefix from the initial state. Since prefixes can be

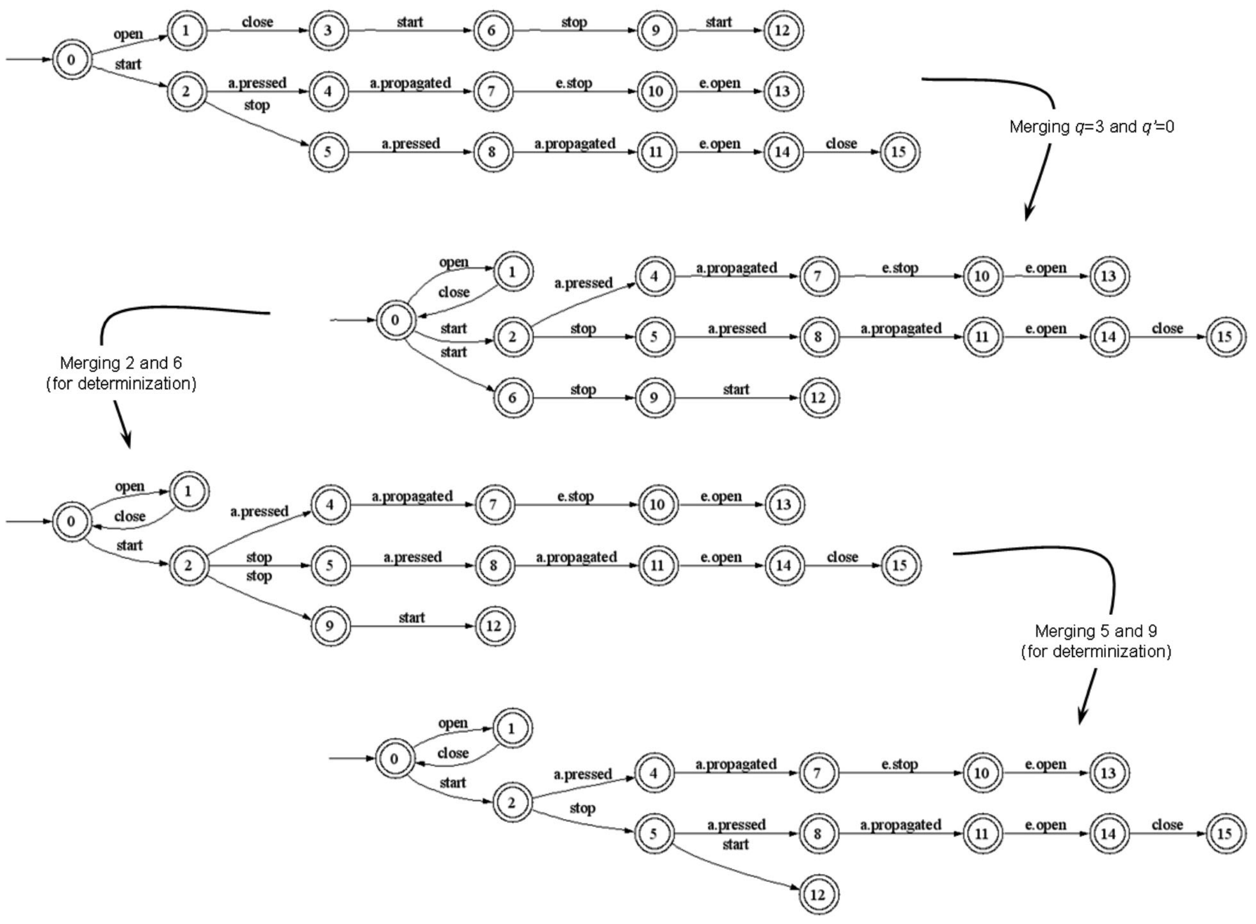

Fig. 12. Typical steps implemented by the *Merge* function. From the current Solution A, states 3 and 0 are merged. The resulting NFA is converted into a deterministic quotient automaton A_{new}. Labels *e.stop*, *e.open*, *a.pressed*, and *a.propagated* are shorthand for *emergency stop*, *emergency open*, *alarm pressed*, and *alarm propagated*, respectively.

sorted according to that order, the states can be ranked accordingly. For example, the PTA states in Fig. 12 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.

(*Merge*) The *Merge* function merges the two states (q, q') selected by the *ChooseStatePairs* function in order to compute a quotient automaton, that is, to generalize the current set of accepted behaviors. In the example of Fig. 12, we assume that states 0, 1, and 2 were previously determined not to be mergeable (through negative scenarios initially submitted or generated scenarios that were rejected by the user). Merging a candidate state pair may produce a nondeterministic automaton. For example, after having merged $q = 3$ and $q' = 0$ in the upper part of Fig. 12, one gets two transitions labeled *start* from state 0, leading to states 2 and 6, respectively. In such a case, the *Merge* function merges the latter states and, recursively, any further pair of states that introduces nondeterminism.

We call the operation of removing nondeterminism through such a recursive merge *determinization*. This operation guarantees that the current solution at any step is a DFA. It does not remove nondeterminism to build an equivalent DFA as standard algorithms [8] since it produces an automaton which may accept a more general language than 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 Fig. 11, the quotient automaton is discarded by the algorithm when it is detected not to be compatible with the negative sample.

(*GenerateQuestion*) 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 questions*. Recall from Section 4.2 that a sample, i.e., a set of available scenarios, is characteristic of a language L, that is, of a set of event sequences accepted by a global LTS, if it contains enough positive and negative information. On one hand, the required positive information is the set of short prefixes $Sp(L)$ which form the shortest histories leading to each system state. This positive information must also

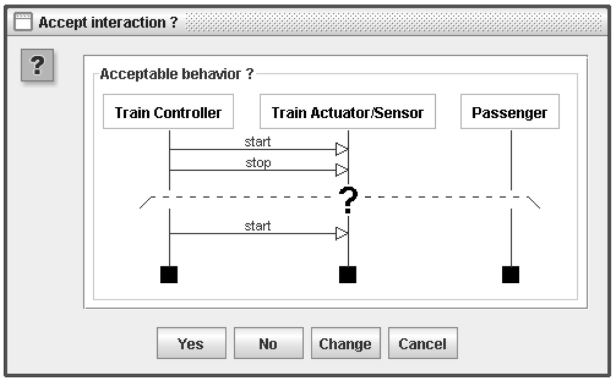

Fig. 13. Scenario question submitted to the end-user by the interactive synthesis algorithm.

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 PTA (as well as any machine generalized from it by merging states) is guaranteed to contain the global LTS states and transitions. 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 which 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.

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, $q' \in Sp(L)$). 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, $q \in N(L)$. 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 deny 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 $xu \in L(A_{new}) \setminus L(A)$. 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 a 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 Fig. 12 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, see Fig. 13, which can be rephrased as follows: *When having started and stopped the train, can the controller restart it again?* One can see that the first episode of this scenario in Fig. 12 is already accepted by A, whereas the entire behavior is accepted in A_{new}.

The suffixes selected by our tool for generating questions are always the entire branches of the tree rooted at q. The aim is to help the end-user to more easily determine whether the generated scenario should be rejected. The boundary between the first (xv) and second (w) episodes of this scenario can be determined by comparing A and A_{new}; some states get new outgoing transitions during the derivation of the quotient automaton—see, e.g., the new transition label *start* appearing on state 5 at the bottom of Fig. 12. Such a new transition identifies the first event of w. Given a selected suffix $u = vw$ of q, v is composed of the transitions folded up during the determinization process, whereas w is the unfolded part of the branch. No scenario has to be generated whenever w is empty, that is, when the entire branch is folded up.

As the formulated behavior is not yet in A but will be in A_{new}, a generated scenario could be viewed as a measure of language generalization by the merging of q and q'. Our interactive RPNI extension controls such generalizations when the sample is not characteristic.

(*CheckWithEndUser*) The end-user is asked to classify generated scenarios as examples of positive or negative behavior of the system; the scenario collection is completed accordingly. For the scenario question in Fig. 13, the user should provide a positive answer, thereby classifying this scenario as a positive example of system behavior.

The compatibility of q and q' in the system LTS gets confirmed when the user classifies as positive all scenarios generated at this step in the algorithm; A_{new} is then considered a good intermediate LTS solution and becomes the new current solution. The algorithm then continues with another state pair.

If one generated scenario is classified as a negative example, the generation procedure ends that solution path. The goal is to avoid merging of incompatible states and one single counterexample is sufficient to avoid such poor generalization. When a scenario gets rejected, the first event in its second episode is used as a prohibited event of the negative example. In case this prohibited event appears later in the second episode, the user may change the position of the boundary between the accepted and rejected episodes, to change the counterexample or make it more specific (see the *Change* button in Fig. 13). In all cases, the generated scenarios are added to the scenario collection once they are classified (see the algorithm in Fig. 11).

The final result of the induction algorithm is guaranteed to be compatible with all scenarios received, including those classified by the end-user. Compatibility with the positive scenarios is ensured by construction of the initial PTA and subsequent merges, which can only generalize the accepted set of behaviors (see Section 2.4). At each step of the algorithm, the new solution is only kept if it rejects all current negative scenarios, as checked by the *Compatible* function, and if all new scenarios are classified as positive by the end-user.

4.3 Projecting the Global System LTS on Agents

Fig. 14a shows the train global LTS obtained by the synthesis algorithm. The LTS for each agent forming that

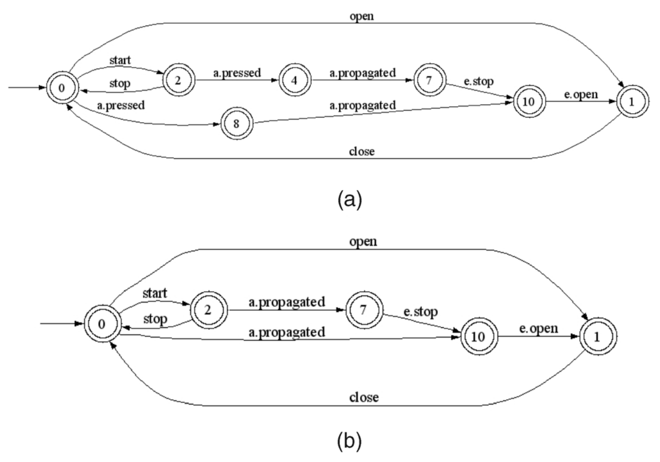

Fig. 14. (a) Synthesized LTS of the global train system. (b) Train Controller LTS.

system are generated from this global LTS using standard automaton algorithms [8].

The projection of the system LTS on a specific agent proceeds in three steps:

1. Each event not monitored or controlled by the agent is replaced by a special empty event ε and eliminated.
2. The resulting NFA is converted into an equivalent DFA.
3. The resulting LTS is minimized to yield the minimal deterministic LTS representing the behavior of this agent.

Fig. 14b shows the resulting LTS for the Train Controller.

An earlier version of our tool was based on an alternative approach where a high-level Message Sequence Chart [10] was generated first as an intermediate product before LTS generation. The generated high-level Message Sequence Chart (hMSC) in this approach has exactly the same admissible behaviors as the synthesized LTS. We initially took that approach for two reasons: 1) The agent LTS can then be generated from this hMSC using known techniques [29] and 2) an intermediate hMSC may sometimes provide a valuable global view of the system for validation by the analyst. The synthesis approach presented in the paper has, however, been preferred for much greater simplicity.

4.4 Discussion

This section addresses three issues raised by our LTS synthesis technique: the adequacy of the synthesized behavior model, the presence of implied scenarios, and the number of scenario questions interactively generated.

4.4.1 Adequacy of the Synthesized Model

Unlike deductive inference, inductive inference from examples is known to be logically not sound. The inferred system model may be undergeneralized or overgeneralized.

Overgeneralization occurs when the synthesized system LTS covers undesired behaviors. This may occur when states were merged by the algorithm while they correspond to distinct states of the system, that is, some system states may not have been adequately identified. Although scenario generation is based on characteristic samples, such situations may arise when the sample is sparse due to a limited number of generated scenario questions. One possible way of fixing this is to add the undesired behaviors being covered as new negative examples to the scenario collection and restart the incremental synthesis algorithm.

Undergeneralization occurs when desired system behaviors are not covered by the system LTS. This may occur when the scenario collection is not structurally complete; some system transitions may not have been adequately identified. One possible way of fixing this is to add the missing desired behaviors as new positive examples to the scenario collection and restart the incremental synthesis algorithm.

The union of the set of initial end-user scenarios, the set of answers to scenarios questions, and the set of such additional desired/undesired examples is converging toward a characteristic sample for the algorithm. Keeping classified scenarios in an updated collection thus contributes to the algorithm's convergence toward an adequate behavior model. It also prevents the same scenario from being resubmitted during the following cycles of the incremental synthesis.

Under and overgeneralizations should be detected before they can be fixed. This may be achieved through model checking [21], [5], model animation [22], or model validation using the LTS decoration algorithm presented in the next section.

4.4.2 Handling Implied Scenarios

The set of behaviors of the parallel composition of each agent LTS is not necessarily the same as the set of behaviors of the synthesized global LTS. Implied scenarios may result from the parallel composition of agents acting on local information [28]. In our example, implied scenario analysis would result in new questions to the user such as: *"Can the passenger push on the alarm twice?"* Nondesired implied scenarios should be detected and excluded. The interested readers may refer to [28] and [20] for details on implied scenario analysis.

4.4.3 Reducing the Number of Submitted Questions

According to the definition of a characteristic sample, our RPNI-based strategy is optimistic; 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 not characteristic for the system LTS.

To overcome this problem, our tool implements an optimized strategy known as *Blue Fringe* [19]. The difference lies in the way state pairs are considered for merging. The general idea is to first consider state pairs for which compatibility has the highest chance being confirmed by the user through positive classification. The resulting "please confirm" interaction may also appear more appealing to the user.

Fig. 15 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). 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 considers the lowest-rank blue state first (state 4 here) for merge with the

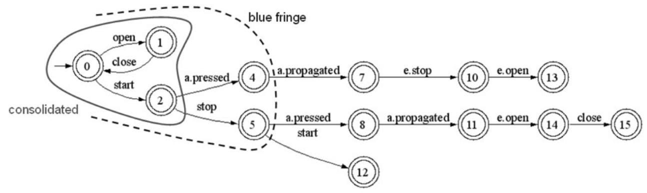

Fig. 15. Consolidated states (red) and states on the fringe (blue) in a temporary solution.

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 questions being generated to avoid poor generalization. Moreover, such questions 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 optimization 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 through an adapted version of the *Compatible* and *Initialize* functions. *Initialize* 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 as positive or negative. The *Compatible* function now returns a compatibility score instead of a Boolean value. The score is defined as -1 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 an 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 questions.

In the train example of this paper, the original RPNI-based algorithm in Section 4.2 learns the system LTS correctly by submitting 20 scenarios to the end-user (17 should be rejected and only three should be accepted). With the interactive Blue-Fringe optimization, the same LTS is synthesized with only three scenarios being submitted (one to be rejected and two to be accepted).

The number of generated questions can be further reduced by integration of system knowledge or legacy components. Domain properties and legacy components can be modeled as state machines. As the states of the PTA capture global system states, defined as tuples of simultaneous LTS states of the various agents, adding partial state information in the PTA allows an equivalence relation to be defined on states. Any generalization which would result in merging nonequivalent states according to this relation can be discarded. This ensures both consistency with system knowledge and search space reduction, which speeds up the learning process and decreases the number of scenario questions. An example of legacy component integration is presented in Section 6. Some preliminary work on integrating goal specifications [16] in the learning process suggests that goals are another highly effective source for drastic pruning of scenario questions.

5 Generating State Invariants from Fluent Definitions

State invariants on a state machine are assertions on a specific state which hold every time that state is visited. The annotation of state machines with such invariants provides multiple benefits:

- The understandability and documentation of the state machine is improved.
- The invariants can be shown to the analyst for validation and error detection.
- State invariants can be used by analysis tools for more efficient analysis [12].
- Invariants can be used by code generators to improve the quality of the generated code. When the entire code of the application cannot be fully generated, the generated fragments must be readable by developers. The choice of variables in the generated code, their names, and property annotations may then be quite useful.

An algorithm for generating state invariants for SCR state machines is presented in [12]. This algorithm cannot be applied here as the semantics of SCR state machines differs from the semantics of LTS state machines. Consider a state machine with state q and no outgoing transition with label l.

- According to the semantics described in [12], if the system is in state q, an event with label l can occur; the system then remains in the same state q.
- With the LTS semantics, no event with label l can occur when the system is in state q.

This section presents an algorithm for generating state invariants from fluent definitions. We use fluents, as introduced in Section 2.4, rather than pre and postconditions on event occurrences. The reasons are the following:

- Fluents turn out to be simpler to use; they allow the analyst to focus on one single state variable at a time and thereby annotate LTS incrementally.
- Invariant generation requires less input information; only the counterpart of postconditions has to be defined, transition preconditions are provided by the state machine.
- Conflicts may arise when the pre/postconditions are incompatible with the state machine. Consider two consecutive events, ev1 and ev2, in an LTS execution. The postcondition post(ev1) and the precondition pre(ev2) cannot be inconsistent; in other words, a conflict arises when

$\mathrm{post(ev1)} \wedge \mathrm{pre(ev2)} \rightarrow \mathrm{false}$, meaning that either the state machine or the pre/postconditions are incorrect.

- Fluents provide a nice interface between goal specifications and goal operationalizations [16]; their definition can be derived in a systematic way from goal specifications provided by goal models.

The algorithm implemented by our tool decorates each state of the input LTS with a conjunction of fluent values that holds in that state.

For our train example, three fluents are emerging from goal specifications:

- fluent *moving* = <{*start*}, {*stop, emergency stop*} > initially *false*,
- fluent *doors_open* = <{*open doors, emergency open*}, {*close doors*}> initially *false*,
- fluent *alarmed* = <{*alarm propagated*}, {*emergency open*}> initially *false*.

The decoration of the initial state will be:

$$\neg\mathrm{moving} \wedge \neg\mathrm{doors\_opened} \wedge \neg\mathrm{alarmed}.$$

Section 5.1 discusses the decoration of LTS states with single fluents. Section 5.2 then shows how state invariants are formed from single decorations.

5.1 Decorating States with Single Fluents

We first discuss how fluent values can decorate a node on a single LTS path. Then, we consider the general case of node decoration for a node pertaining to multiple paths.

5.1.1 Fluent Values for a Node on a Single Path

A fluent Fl is true after a finite LTS execution σ ending in state q if and only if one of the following conditions holds [5]:

1. Fl holds initially and no terminating event has occurred in σ.
2. Some initiating event has occurred in σ with no terminating event occurring since then.

The following recursive version of this definition is useful for our generation algorithm.

A Fluent Fl is true after a finite LTS execution σ ending in state q if and only if one of the following conditions holds:

1. σ is empty and Fl holds initially.
2. The last event of σ belongs to the initiating events.
3. The last event e of σ does not belong to the terminating events and the fluent is true after σ', where $\sigma = <\sigma', e>$.

5.1.2 Fluent Values for a Node on Multiple Paths

The value of a fluent in an LTS node is not necessarily either true or false. There may be two LTS executions σ_1 and σ_2 reaching a state q such that the value of the fluent Fl after σ_1 is true and the value of the fluent Fl after σ_2 is false. For example, consider the train controller LTS (Fig. 10) and the fluent

$$emergency =< alarm\ propagated, start > initially\ false.$$

The value of the fluent at the initial state after the LTS execution $\sigma_1 =< start, stop >$ is false. The value of the fluent at the initial state after the LTS execution $\sigma_2 =< alarm\ propagated, emergency\ open, close\ doors >$ is $true$. The value of *emergency* at that state will therefore be set to *top*, meaning that the value of the fluent can be true or false at that state depending on the path being followed. It is also possible that a state is unreachable from the initial state. In that case, the value of Fl at that state will be set to *bottom*. The possible values of a fluent in some state thus belong to a lattice $Bool_{\mathrm{abs}}$ defined as follows:

$$\mathrm{Bool_{abs}}: \begin{tikzcd}[row sep=small] & \text{top} & \\ \text{true} \arrow[ur, dash] \arrow[dr, dash] & & \text{false} \arrow[ul, dash] \arrow[dl, dash] \\ & \text{bottom} & \end{tikzcd}$$

```
if decor(source) ≠ bottom then
    if $event \in Init_{Fl}$ then
        $newVal \leftarrow$ true
    else
        if $event \in Term_{Fl}$ then
            $newVal \leftarrow$ false
        else
            $newVal \leftarrow$ decor(source)
    decor(target) ← sup(decor(target), newVal)
```

Fig. 16. Propagation rule of the decoration algorithm.

Given an LTS model (Q, Σ, δ, q_0) and a fluent $Fl =< Init_{Fl}, Term_{Fl} > initially\ Initially_{Fl}$, the output of our algorithm is a function decor: $Q \rightarrow bool_{abs}$ defined as follows:

$decor(q) =$

'true'	iff Fl is true for all LTS executions reaching state q
'false'	iff Fl is false for all LTS executions reaching state q
'top'	iff Fl is true for some LTS executions reaching state q and false for others
'bottom'	iff state q is not reachable from the initial state.

The idea behind the algorithm is the following: At each step, every state has a decoration. At the beginning, in every state, the fluent value is initialized to *bottom* except for the initial state where the fluent is initialized to the initial value provided by its definition. The algorithm propagates fluent values in the state machine according to a rule derived from the recursive characterization of these values. This propagation rule is shown in Fig. 16, where $source \in Q$, $event \in \Sigma$, $target \in \delta(source, event)$, and sup is the supremum function in the lattice $Bool_{abs}$, for instance, $sup(true, false) = top$ and $sup(bottom, true) = true$.

The algorithm applies this propagation rule for each transition until a fixpoint is reached where no state decoration changes if the propagation rule is applied on each transition once again.

The algorithm for one fluent is given in Fig. 17. It keeps track of the set *ToExpl* of states that have been updated and where the propagation should be applied. The algorithm stops when the set *ToExpl* is empty, that is, there is no state that must still propagate its value.

The algorithm terminates because the set *ToExpl* will eventually be empty as a state can change his decoration at

```
Input: A LTS $(Q, \Sigma, \delta, q_0)$
       A fluent $Fl = \langle Init_{Fl}, Term_{Fl} \rangle$ initially $Initially_{Fl}$
Output: decor: $Q \rightarrow bool_{abs}$            //A decoration function
foreach $q \in Q$ do
    decor($q$) $\leftarrow$ bottom
$ToExpl \leftarrow \{q_0\}$
decor($q_0$) $\leftarrow Initially_{Fl}$
while $ToExpl \neq \emptyset$ do
    $source \leftarrow$ getOne ($ToExpl$)
    $ToExpl \leftarrow ToExpl \setminus \{source\}$
    foreach $(target, event)$ such that $target \in \delta(source, event)$ do
        if $event \in Init_{Fl}$ then
            $newVal \leftarrow$ true
        else
            if $event \in Term_{Fl}$ then
                $newVal \leftarrow$ false
            else
                $newVal \leftarrow$decor($source$)
        $newDecor \leftarrow$ sup(decor($target$), $newVal$)
        if $newDecor \neq$ decor($target$) then
            $ToExpl \leftarrow ToExpl \cup \{target\}$
        decor($target$)$\leftarrow newDecor$
return decor
```

Fig. 17. LTS decoration algorithm.

most twice: from bottom to either true or false and from true or false to top. The reason is that the lattice is finite and the decoration can only go up in the lattice in view of the supremum operator.

Fig. 18 illustrates the two first steps of our algorithm for the fluent *moving* = <{*start*}, {*stop, emergency stop*}> initially false on the *Train Controller* state machine. The value of the fluent at each state and after each step is shown in Table 1.

(Step 0) We initialize the values of *moving* at *bottom* for each state except for the initial state where we initialize *moving* to its initial value.

(Step 1) Only state 0 is in *ToExpl*. We propagate its value to all his successors states. It has three outgoing transitions toward state1, state3, and state4. The fluent value of state0 and state2 thus remains the same. For state1, the *newVal* equals true because *start* belongs to the initiating events and therefore *moving* will equal $sup(bottom, true) = true$. Therefore, we add state1 to the set *ToExpl*. For state3 and state4, the events *alarm propagated* and *open doors* do not belong to the initiating or terminating events. The fluent values of state3 and state4 are $sup(bottom, false) = false$. We thus add state3 and state4 to *ToExpl*.

(Step 2) We choose one element in *ToExpl*. Here, we choose state1. State1 has two outgoing transitions. Because *stop* is a terminating event, *moving* at state0 equals $sup(false, false) = false$. Because the decoration of state0 has not changed, we do not add state0 to *ToExpl*. *Alarm propagated* does not belong to the initiating or terminating events and, therefore, *moving* at state2 equals $sup(bottom, true) = true$. We thus add state2 to ToExpl.

The process is continued until *ToExpl* is empty.

5.2 Forming the State Invariant

A straightforward way of obtaining the global invariant at each state is to apply the algorithm of Fig. 16 once per fluent and then take the conjunction of results. For example, the fluent values for the initial state are

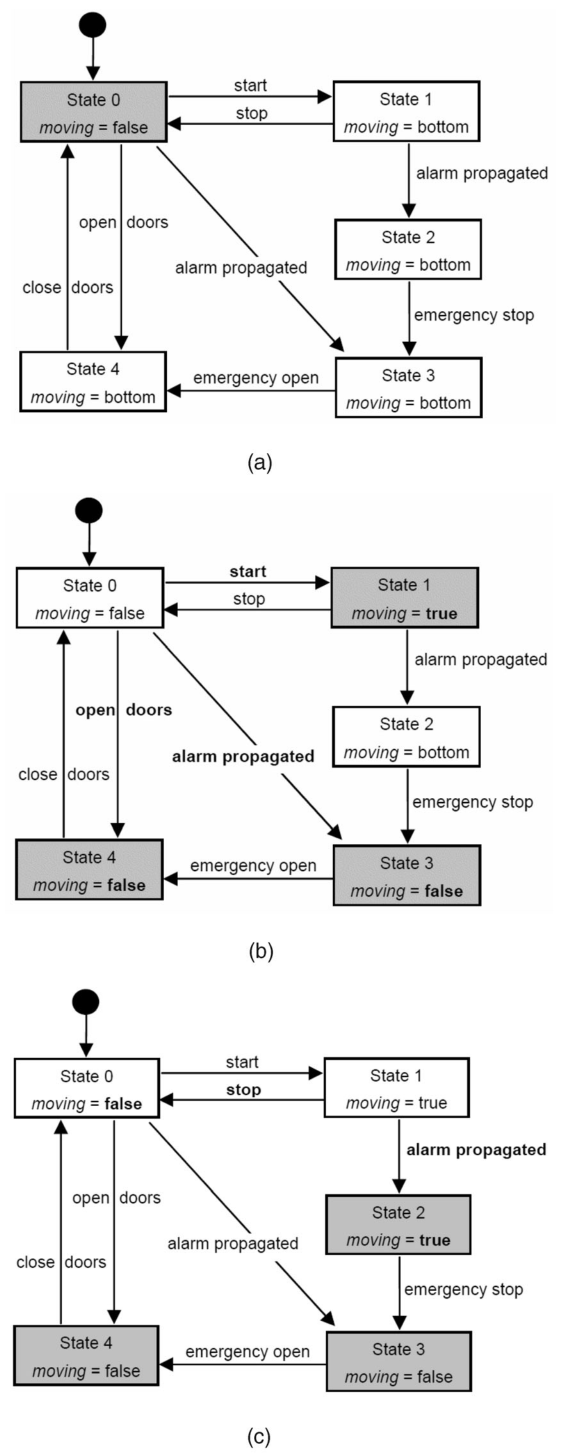

Fig. 18. Executing the algorithm for fluent moving. (a) After initialization (Step 0). (b) After having propagated the value of fluent at state0 (after Step 1). (c) After having propagated the value of fluent at state1 (after Step 2).

$$\{\text{moving} = \text{false}, \text{doors\_opened} = \text{false}, \text{alarmed} = \text{false}\}.$$

The state invariant is formed by taking the conjunction of all fluent values in that state. The *top* values do not appear in this conjunction as they do not give any information about the values of the corresponding fluents. The state invariant at the initial state is thus

$$\neg\text{moving} \wedge \neg\text{doors\_opened} \wedge \neg\text{alarmed}.$$

TABLE 1
Values of Fluent Moving after Each Step of the Decoration Algorithm

Step	ToExpl	State0	State1	State2	State3	State4
		false	bottom	bottom	bottom	bottom
1	{**0**}	false	**true**	bottom	**false**	**false**
2	{**1**,3,4}	**false**	true	**true**	false	false
3	{**3**,4,2}	false	true	true	false	**false**
4	{**4**,2}	**false**	true	true	false	false
5	{**2**}	false	true	true	**false**	false

The algorithm implemented in our tool is an equivalent, optimized version where the value of all fluents is calculated within a single loop.

6 Case Study: A Mine Pump Control System

This section shows our tool in action on a nontrivial benchmark. We consider the following simplified problem statement for the Mine Pump exemplar [13]: "*Water percolating into a mine is collected in a sump to be pumped out of the mine. The water level sensor detects when water is above and below a specific level. A pump controller switches the pump on when the water goes above this level and off when it goes below this level. To avoid the risk of explosion, the pump must be operated only when the methane level is below some critical level.*"

Fig. 19 shows three scenarios initially submitted to the tool by stakeholders—two positive and one negative. As mentioned before, the tool has the built-in assumption that all submitted input scenarios start in the same initial state (this assumption is commonly made by the other approaches to state machine synthesis from scenarios.) The submitted scenarios thus implicitly specify that both the water level and the methane level are initially low.

To illustrate the integration of legacy components, we assume that the methane sensor is an external component specified by the LTS shown in Fig. 20.

From this input, the tool generates a first scenario question, shown in Fig. 21a. This scenario should be rejected by the end-user; in view of the common initial state with low water, the pump controller may not switch the pump on when the water level is low. This scenario is added as a negative scenario to the initial collection. Its precondition contains the events signal critical methane and signal not critical methane and its prohibited event is switch pump on.

The next scenario then generated by the tool is shown in Fig. 21b. This scenario is to be accepted by the user; if the water level is high and the pump is running, the controller should switch the pump off when the methane level becomes critical and should switch the pump on when the methane level is no longer critical.

The third and last scenario question requires a negative answer. The synthesis already terminates after those three questions as the integration of the legacy component allows the generation of two scenarios violating the methane sensor LTS to be avoided. Fig. 22 shows the synthesized system LTS.

The tool then projects the synthesized system LTS on each agent. Fig. 23 shows the LTS for the pump controller, where the predicates annotating nodes there are to be replaced by numbers.

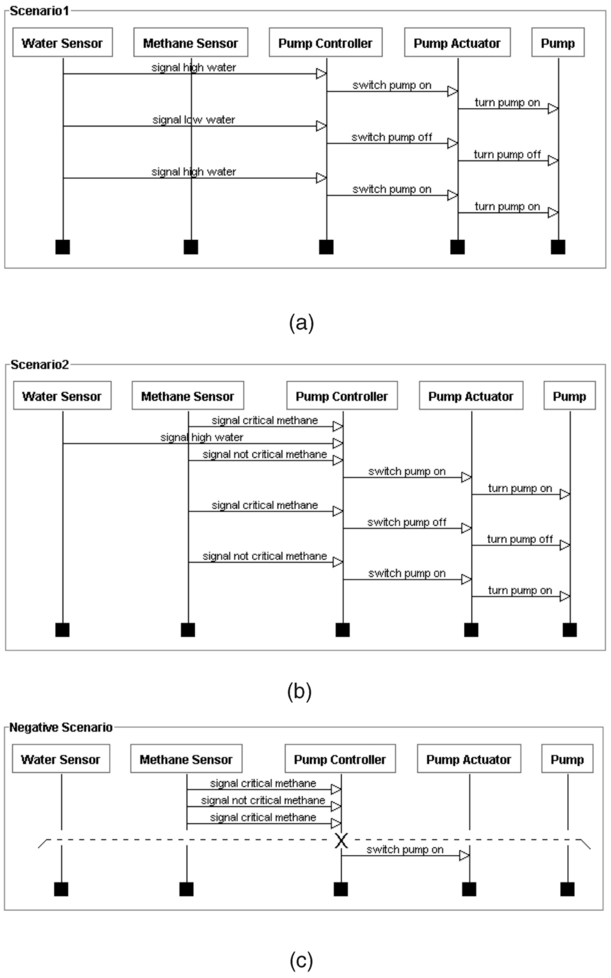

Fig. 19. Three initial scenarios of the mine pump.

The analyst enters the picture at this point. For better understanding of the generated LTS and its validation, she may enter the following fluent definitions as input to the invariant generation process:

- fluent *PumpOn* = <{*turn pump on*}, {*turn pump off*} > initially *false*.
- fluent *HighWater* = <{*signal high water*}, {*signal low water*}> initially *false*.
- fluent *CriticalMethane* = <{*signal critical methane*}, {*signal not critical methane*}> initially *false*.

These fluents define the state of the pump, of the water level, and of the methane level in terms of their initiating/terminating events. They are typically identified from goal operationalizations [16].

The invariant generator then decorates each agent LTS with node annotations as shown for the pump controller LTS in Fig. 23.

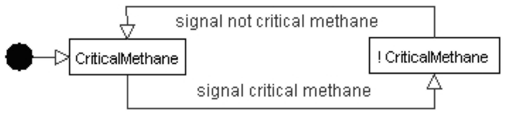

Fig. 20. LTS of the methane sensor.

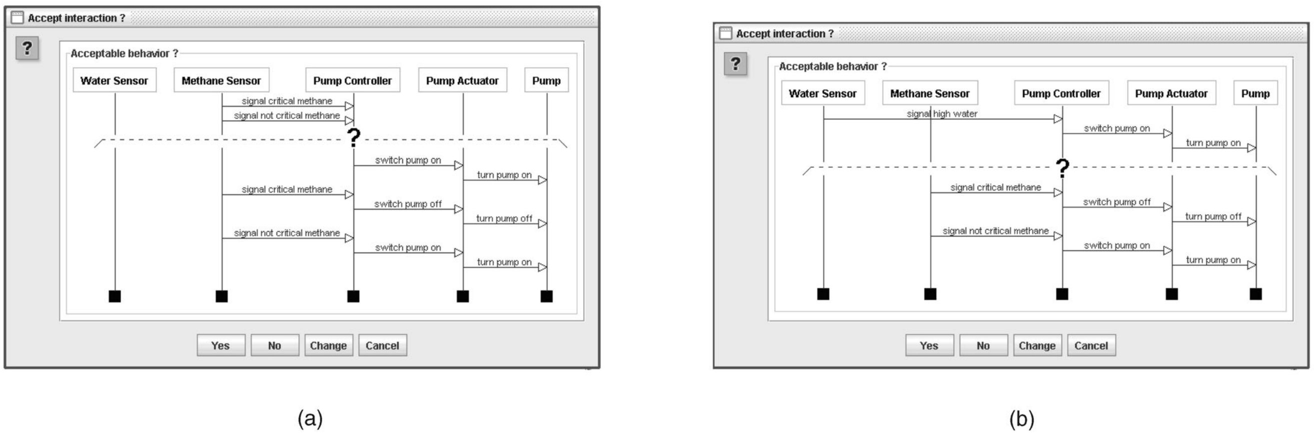

Fig. 21. Two scenarios generated by the tool.

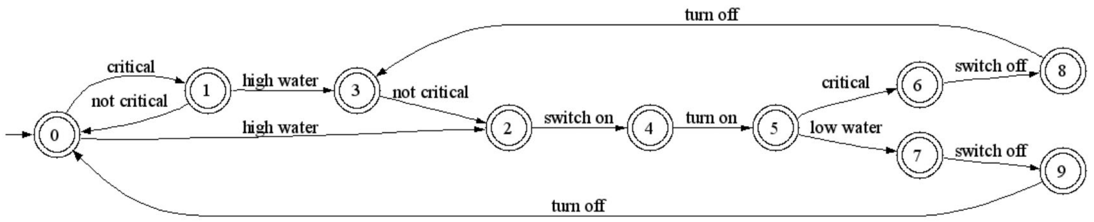

Fig. 22. Synthesized system LTS for the mine pump. Event labels are shorthand for the events presented in the scenarios of Fig. 19.

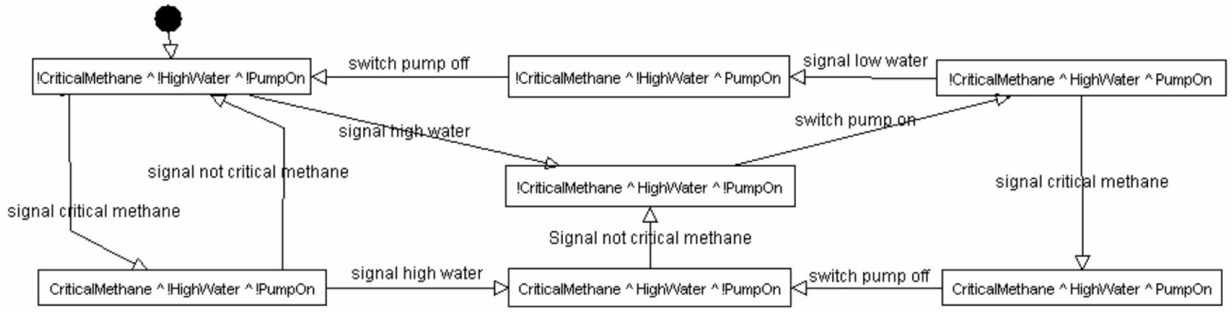
!CriticalMethane ^ !HighWater ^ !PumpOn
switch pump off
!CriticalMethane ^ !HighWater ^ PumpOn
signal low water
!CriticalMethane ^ HighWater ^ PumpOn
signal high water
switch pump on
signal not critical methane
!CriticalMethane ^ HighWater ^ !PumpOn
signal critical methane
signal critical methane
Signal not critical methane
CriticalMethane ^ !HighWater ^ !PumpOn
signal high water
CriticalMethane ^ HighWater ^ !PumpOn
switch pump off
CriticalMethane ^ HighWater ^ PumpOn

Fig. 23. Synthesized annotated LTS of the pump controller.

7 Conclusion

Scenarios are concrete vehicles frequently used for eliciting, illustrating, and validating system requirements and design models. However, they are just partial examples of desired or undesired behavior. The required properties are left implicit and must be inferred. The behavior models are only partially covered and need to be synthesized.

The contribution of this paper is twofold.

- A new approach has been presented for synthesizing behavior models from simple forms of scenarios. Compared with other synthesis approaches, our approach does not require additional input to the synthesis process, such as state assertions along episodes or flowcharts on such episodes. Positive and negative scenarios are both taken into account. New scenarios are generated as questions for further elicitation and for control over the generalization process. The synthesis process is incremental and does not require all scenarios to be provided from the beginning. The global system model is guaranteed to cover all positive scenarios and exclude all negative ones, whereas the projection of this model on system agents may introduce, as in other approaches, undesirable implied scenarios that result from the parallel composition of the system agents.
- A new fixpoint algorithm has been described for generating state invariants on labeled transition systems so as to make such models easier to understand and validate—especially as they are automatically generated from scenarios.

The tool implementing both contributions has been used on several nontrivial case studies and example benchmarks from the literature; it proved to be quite effective in synthesizing LTS models through scenario-based interaction only. For example, a state machine for an ATM system is generated in [31] from four scenarios annotated with pre and postconditions. Our tool generates the same state machine from the same four input scenarios, but without any annotation, with only one scenario question (to be rejected by the user). In [23], the MAS tool takes one single scenario for an alarm clock system as input. MAS first interacts four times with the user through trace questions on the state machine generated for the so-called control unit agent. The resulting state machine appears to be overgeneralized; the user therefore must further submit a counterexample. Three more trace interactions are then required before the final state machine is generated. On this clock example with the same input scenario, our tool generates one single scenario question. This question corresponds to the counterexample that the user had to provide by herself in [23]. We generate it and express it in a simple MSC form. Since the single input scenario is not structurally complete, the state machine generated by our tool is not complete. Adding one more positive scenario makes our tool generate the same state machine as in [23] without any further question. In addition, the tool generates the state machines for each agent.

Our LTS synthesis technique extends a known learning algorithm from the literature on grammatical inference [26], [4] to make it incremental and generate scenario questions. The convergence of this algorithm is guaranteed when the learning sample is rich enough [27]—said in scenario terms, when the scenario collection is characteristic. This shows the importance of positive as well as negative scenarios. The positive scenarios help in learning a system LTS by providing a constraint on minimal coverage of behaviors; on the other hand, the negative scenarios help in learning the correct system by constraining such coverage not to be too wide. Our extension of this algorithm is aimed at overcoming the limitation of not having a characteristic scenario collection from the beginning. Additional scenarios are generated during LTS synthesis to be classified by the end-user as positive or negative. Those scenario questions are generated so as to converge toward a characteristic scenario sample for the learned system. This ensures convergence of the learning process while providing a natural way of eliciting further scenarios and their underlying requirements. As discussed in Section 4.4 and Section 6, various optimizations of this interactive induction algorithm help reduce the number of scenario questions without sacrificing its convergence.

Our approach raises several issues. Multiple users may submit inconsistent scenarios. As with other approaches, inconsistencies may propagate to the synthesized model. Our approach is also highly sensitive to classification errors. Accepting a scenario question instead of rejecting it, for example, will obviously result in inadequate models being synthesized (this problem is common to learning-by-examples techniques).

Our tool makes the assumption that all submitted scenarios start in the same initial system state (this assumption is used in other approaches as well). For real-sized systems or multiuser elicitation, such an assumption seems unrealistic. If two end-users specify scenarios separately, they might not choose the same initial state. Our algorithm should be adapted to support this problem.

The generated state machines are "flat" LTS. Transforming them into hierarchical machines with sequential and parallel decomposition within single agents would be useful, in particular, for generation of more structured code. The generated fluent decorations might help in such a structuring process.

In addition to existential scenarios, we would like to explore the use of universal scenarios [2] in our approach. Existential scenarios illustrate what may occur; universal scenarios state what must occur. The addition of universal scenarios would allow us to constrain the induction process, automatically reject induced solutions that are incompatible with the information provided by obligations, and thus further reduce the number of scenario questions.

We are currently experimenting with our approach to synthesize behavior models for Web applications from end-user scenarios of interaction with such applications. It turns out that the state space gets much larger in this context, which may result in too many scenario questions being generated to the end-user. Constraining the search space through additional input information is one obvious solution to this problem. Such additional information, to be provided by the analyst (not the end-user), may include LTS models of legacy components to be integrated (as already suggested in Section 6). The latest version of our tool also takes system goals [16], to be achieved by the synthesized models, to further constrain the search space. Preliminary experiments suggest some drastic improvement. This is not too unexpected; as a goal captures a set of scenarios, the number of scenarios questions decreases accordingly.

Acknowledgments

This work was partially supported by the Regional Government of Wallonia (ReQuest project, RW Conv. 315592). Thanks are due to Renaud De Landtsheer for his help in constructing the algorithm for generating state invariants from fluent definitions and to Emmanuel Letier for helpful comments and discussions. The authors are also grateful to the reviewers for helpful comments which hopefully resulted in the clarification of several aspects of our synthesis technique.

References

[1] R. Alur and M. Yannakakis, "Model Checking of Message Sequence Charts," *Proc. 10th Int'l Conf. Concurrency Theory*, pp. 114-129, 1999.

[2] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, no. 1, 2001.

[3] P. Dupont, L. Miclet, and E. Vidal, "What Is the Search Space of Regular Inference?" *Grammatical Inference and Applications*, pp. 199-210, 1994.

[4] P. Dupont, "Incremental Regular Inference," *Grammatical Inference: Learning Syntax from Sentences*, pp. 222-237, 1996.

[5] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems," *Proc. ESEC/FSE 2003*, 2003.

[6] E.M. Gold, "Language Identification in the Limit," *Information and Control*, vol. 10, no. 5, pp. 447-474, 1967.

[7] E.M. Gold, "Complexity of Automaton Identification Data," *Information and Control*, vol. 37, pp. 302-320, 1978.

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

[9] H. Ichikawa, M. Itoh, J. Kato, A. Takura, and M. Shibasaki, "SDE: Incremental Specification and Development of Communications Software," *IEEE Trans. Computers*, vol. 40, pp. 553-561, 1991.

[10] ITU, *Message Sequence Charts, Recommendation Z.120*, Int'l Telecom Union, Telecomm. Standardization Sector, 1996.

[11] *IEEE Trans. Sofware Eng.*, special issue on scenario management, M. Jarke and R. Kurki-Suonio, eds., vol. 24, no. 12, Dec. 1998.

[12] R. Jeffords and C. Heitmeyer, "Automatic Generation of State Invariants from Requirements Specifications," *Proc. Sixth ACM Symp. Foundations of Software Eng.*, 1998.

[13] M. Joseph, *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall Int'l, 1996.

[14] J. Kramer et al., "CONIC: An Integrated Approach to Distributed Computer Control Systems," *IEE Proc.*, Part E 130, 1, pp. 1-10, Jan. 1983.

[15] I. Kruger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," *Proc. IFIP Wg10.3/Wg10.5 Int'l Workshop Distributed and Parallel Embedded Systems,* F.J. Rammig, ed., pp. 61-71, 1998.

[16] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," *Proc. RE'01—Fifth Int'l Symp. Requirements Eng.,* pp. 249-263, Aug. 2001.

[17] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios," *IEEE Trans. Sofware. Eng.,* vol. 24, no. 12, Dec. 1998.

[18] K.J. Lang, "Random DFAs Can Be Approximately Learned from Sparse Uniform Examples," *Proc. Fifth ACM Workshop Computational Learning Theory,* pp. 45-52, 1992.

[19] K.J. Lang, B.A. Pearlmutter, and R.A. Price, "Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm," *Grammatical Inference,* pp. 1-12, Springer-Verlag, 1998.

[20] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Monitoring and Control in Scenario-Based Requirements Analysis," *Proc. ICSE 2005—27th Int'l Conf. Software Eng.,* May 2005.

[21] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs.* Wiley, 1999.

[22] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer, "Graphical Animation of Behavior Models," *Proc. ICSE 2000: 22nd Int'l Conf. Software Eng.,* pp. 499-508, May 2000.

[23] E. Mäkinen and T. Systä, "MAS—An Interactive Synthesizer to Support Behavioral Modelling in UML," *Proc. ICSE 2001—Int'l Conf. Software Eng.,* May 2001.

[24] R. Milner, *Communication and Concurrency.* Prentice-Hall, 1989.

[25] R. Miller and M. Shanahan, "The Event Calculus in Classical Logic—Alternative Axiomatisations," *Linkoping Electronic Articles in Computer and Information Science,* vol. 4, no. 16, pp. 1-27, 1999.

[26] J. Oncina and P. García, "Inferring Regular Languages in Polynomial Update Time," *Pattern Recognition and Image Analysis,* N. Perez de la Blanca, A. Sanfeliu, and E. Vidal, eds. pp. 49-61, World Scientific, 1992.

[27] J. Oncina, P. García, and E. Vidal, "Learning Subsequential Transducers for Pattern Recognition Interpretation Tasks," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 15, no. 5, pp. 448-458, May 1993.

[28] S. Uchitel, J. Kramer, and J. Magee, "Detecting Implied Scenarios in Message Sequence Chart Specifications," *Proc. ESEC/FSE'01—Ninth European Software Eng. Conf. & ACM SIGSOFT Symp. Foundations of Software Eng.,* Sept. 2001.

[29] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of Behavioral Models from Scenarios," *IEEE Trans. Software Eng.,* vol. 29, no. 2, pp. 99-115, Feb. 2003.

[30] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenario Usage in System Development: A Report on Current Practice," *IEEE Software,* Mar. 1998.

[31] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios," *Proc. ICSE 2000: 22nd Int'l Conf. Software Eng.,* pp. 314-323, 2000.

Christophe Damas received the degree of engineer in computer science in 2003 from the Université Catholique de Louvain, Belgium. He is currently pursuing research on generating Web applications from end-user scenarios.

Bernard Lambeau received the degree of engineer in computer science in 2003 from the Université Catholique de Louvain, Belgium. He is currently pursuing research on generating Web applications from end-user scenarios.

Pierre Dupont received the MS degree in electrical engineering from the Université Catholique de Louvain, Belgium, in 1988 and the PhD degree in computer science from l'Ecole Nationale Supérieure des Télécommunications, Paris, in 1996. From 1988 to 1991, he was a research staff member at the Philips Research Laboratory Belgium. In 1992, he joined the France Telecom Research Center, Lannion, France. Its primary research was in automatic speech recognition with a special focus on search algorithms and language modeling. He was a visiting researcher at Carnegie Mellon University, Pittburgh in 1996-1997 and at the Universidad Politécnica de Valencia, Spain, in 1994, 1995, and 2000. From 1997-2001, he was an associate professor in the Computer Science Department of the Université Jean Monnet, Saint-Etienne, France. He is currently a professor in the Department of Computing Science and Engineering and cofounder of the Machine Learning Group of the Université Catholique de Louvain, Belgium. His current research interests include grammar and automata induction, statistical language modeling, graph mining, machine learning applications to natural language processing, and computational biology.

Axel van Lamsweerde is a full professor in the Department of Computing Science of the Université Catholique de Louvain, Belgium. He has been a research associate at Philips Research Labs and a professor at the Universities of Namur, Brussels, and Louvain, Belgium. He has also been a research fellow at the University of Oregon and the Computer Science Laboratory of SRI International, Menlo Park, California. He was a cofounder of two software technology transfer centers supported by the European Union. His research interests are in precise techniques for requirements engineering, system modeling, high assurance systems, lightweight formal methods, process engineering, and knowledge-based software development environments. Since 1991, he has been instrumental in the development and transfer of the KAOS goal-oriented requirements engineering language, method, and toolset. He has been editor-in-chief of the *ACM Transactions in Software Engineering and Methodology* (*TOSEM*), program chair of major international software engineering conferences, including ESEC '91 and ICSE '94, and a founding member of the IFIP WG2.9 Working Group on Requirements Engineering. He was the invited keynote speaker at ICSE 2000. He became an ACM fellow in 2000 and received the ACM Sigsoft Distinguished Service Award in 2003. His recent papers can be found at http://www.info.ucl.ac.be/people/avl.