Scenarios, Goals, and State Machines: a Win-Win Partnership for Model Synthesis

Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde

Dept. Ingénierie Informatique, Université catholique de Louvain
B-1348 Louvain-La-Neuve (Belgium)
{damas, blambeau, avl}@info.ucl.ac.be

ABSTRACT

Models are increasingly recognized as an effective means for elaborating requirements and exploring designs. For complex systems, model building is far from an easy task. Efforts were therefore recently made to automate parts of this process, notably, by synthesizing behavior models from scenarios of interactions between the software-to-be and its environment. In particular, our previous interactive synthesizer generates labelled transition systems (LTS) from simple message sequence charts (MSC) provided by end-users. Compared with others, the synthesizer requires no additional input such as state or flowcharting information. User interactions consist in simple scenarios generated by the synthesizer that the user has to classify as example or counterexample of desired behavior.

Experience with this approach showed that the number of such scenario questions may become fairly large in interaction-intensive applications such as web applications. In this paper, we extend our model synthesis technique by injecting additional information into the synthesizer, when available, in order to constrain induction and prune the inductive search space. Additional information may include global definitions of fluents that link interaction events and atomic assertions; declarative properties of the domain; behavior models of external components; and goals that the software system is expected to satisfy. We provide comparative data on increasingly complex examples to show how effective such constraints are in reducing the number of scenario questions and in increasing the adequacy of the synthesized model. As goals and domain properties might not be easily provided by users, the paper also shows how our synthesizer generates a significant class of them automatically from the available scenarios. As a side-effect, our work provides additional evidence on the synergistic links between scenarios, goals, and state machines for model-driven engineering of requirements and designs.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification - *methodologies, languages, tools*.

General Terms: Design, Languages, Verification.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

Keywords

Synthesis of behavior models, goal-oriented requirements engineering, scenario-based elicitation, scenario generation, labelled transition systems, message sequence charts, incremental learning, analysis tools.

1 INTRODUCTION

Model-driven elaboration, validation, and documentation of requirements and designs call for rich models of the system-to-be. Such models need to cover the intentional, structural, and behavioral dimensions of the system [11] – by *system* we mean both the target software and its environment. Goals, scenarios, and state machines form a golden triangle along the intentional and behavioral dimensions.

- *Goals* are prescriptive statements of intent whose satisfaction requires cooperation among the agents forming the system. Goal models are AND/OR graphs that capture how functional and non-functional goals contribute positively or negatively to each other. Such models support various forms of early, declarative, and incremental reasoning for, e.g., goal refinement and completeness checking, conflict management, hazard analysis, threat analysis, requirements document generation, and so forth [12]. On the down side, goals are sometimes felt too abstract by stakeholders. They cover classes of intended behaviors but such behaviors are left implicit. Goals may also be hard to elicit in the first place and make fully precise.
- *Scenarios* capture typical examples or counterexamples of system behavior through sequences of interactions among agents. They support an informal, narrative, and concrete style of description. Scenarios are therefore easily accessible to stakeholders involved in the requirements engineering process [6]. On the down side, scenarios are inherently partial and cover few behaviors of specific instances. They leave intended system properties implicit. Scenarios may also entail premature design decisions about event sequencing and distribution of responsibilities among system agents.
- *State machines* capture classes of required agent behaviors in terms of states and events firing transitions. They provide visual abstractions of explicit behaviors for any agent instance in some corresponding class. State machines can be composed sequentially and in parallel, and are executable. They can be validated through animation and verified against declarative properties. State machines also provide a good basis for code generation. On the down side, state machines are too operational in the early stages of requirements elaboration. Their elaboration may turn to be quite hard – the diversity of published semantics of state machine formalisms making things even worse [19].

In any case, building rich models that cover the intentional, structural, and behavioral dimensions is a complex process. Automated support is therefore needed.

Recent efforts have been made along this way. For example, a labelled transition system (LTS) model can be synthesized from message sequence charts (MSC) taken as positive examples of system behavior [22]. UML state diagrams can be generated from sequence diagrams capturing positive scenarios [23, 16]. In a similar spirit, MSC specifications can be translated into statecharts [8]. Goal specifications in linear temporal logic can also be inferred inductively from MSC scenarios taken as positive or negative examples [10].

Those techniques all require additional input information beside scenarios, namely: a high-level message sequence chart (hMSC) showing how MSC scenarios are to be flowcharted [22]; pre- and post-conditions of interactions, expressed on global state variables [10, 23]; local MSC conditions [8]; or state machine traces local to some specific agent [16]. Such additional input information may be hard to get from end-users, and may need to be refactored in non-trivial ways in case new positive or negative scenario examples are provided later in the requirements/design engineering process [15].

To address this problem, we have developed a synthesis technique that requires no such additional input information [1]. A global system LTS is synthesized first and then projected on local LTS for each system agent. The global system LTS covers all positive MSCs provided and excludes all negative ones. It is inductively generated through an interactive procedure that extends learning techniques used for grammar induction [20]. In our extension, simple MSC scenarios are generated during synthesis as *questions* to the end-user; the latter just needs to accept or reject the generated scenario and the synthesis continues with the answer being added to the scenario collection. The induction procedure is incremental on training examples.

There is a price to pay with this technique though. While interaction takes place in terms of simple end-user scenarios, and scenarios only, the number of scenario questions may sometimes become large for interaction-intensive applications with complex composite states – as we experienced it when applying the technique to non-trivial web applications.

This paper describes various extensions we implemented in our LTS synthesizer to meet three new objectives:

(a) reduce the number of scenario questions significantly,

(b) produce a more adequate LTS model, that is, a model consistent with knowledge about the domain and about the goals of the target system, and

(c) produce the additional information required for (a) and (b) systematically and, when possible, automatically.

The general principle underlying our extensions is to constrain the induction process in order to prune the inductive search space and, accordingly, the set of scenario questions. The constraints considered in the paper include the following:

- local state assertions along agent timelines;
- LTS models of external components;
- safety properties that capture domain descriptions or system goals.

Constraints of the first type are generated from fluent definitions that link events and atomic predicates [3]. Constraints of the second type are used when they are available from external sources or analyst intervention. Constraints of the third type are obtained in two ways as assertions in fluent temporal logic: (a) systematically, as explanation of why a scenario is rejected as counterexample; (b) automatically, by inference from positive scenarios. In the latter case, the inferred property has to be validated by the user. If it turns to be inadequate, the user is asked to provide a counterexample scenario which will enrich the scenario collection.

The paper is organized as follows. To make the paper self-contained, Section 2 presents some minimal background on scenarios, labelled transition systems, fluent-based specification of goals and domain properties, and our unoptimized LTS synthesis technique. Section 3 shows how the induction process can be constrained through generated state assertions, LTS specifications of external components, domain descriptions, and goal specifications. Section 4 shows how the latter properties can be inferred from scenarios systematically or even automatically. Section 5 evaluates the improvement on our previous technique through comparative data from case studies of varying complexity.

2 BACKGROUND

A simple train system fragment will be used throughout the paper as a running example. The scenarios involve three agent instances: a train controller, a train actuator/sensor, and a passenger. 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 emergency. When the train is moving and the passenger presses the alarm button, the controller must stop the train first and then open the doors in emergency.

2.1 Scenarios as Message Sequence Charts

To represent scenarios, our MSCs are composed of vertical lines representing timelines associated with agent instances, and horizontal arrows representing interactions among them. A timeline label specifies the class of the corresponding agent instance. An arrow label specifies some event defining the corresponding interaction. The event is synchronously produced by the source agent and monitored by the target agent.

A MSC timeline defines a total ordering on incoming/outgoing events. An entire MSC defines a partial ordering on all events. To allow scenario submission by end-users we restrict our MSCs to this very simple form, leaving aside more sophisticated MSC features such as conditions, timers, coregions, etc.

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.

The semantics of MSCs used in this paper is the one introduced in [22]. It is defined in terms of labelled transition systems and parallel composition, see Section 2.2 hereafter. The intuitive, end-user semantics of two consecutive events along a MSC timeline is that the first is *directly* followed by the second.

Fig.1 presents typical scenarios for the train example – three positive and one negative. The labels *e.stop*, *e.open*, *a.pres* and *a.prop* there are shorthands for *emergency stop*, *emergency open*, *alarm pressed*, and *alarm propagated*, respectively. Fig. 1.a shows a positive MSC for the following scenario: "*The train is started by the controller. A passenger then presses the alarm button. The alarm is then propagated to the controller. The latter then stops the train and opens the doors in emergency*". Fig.1.b shows a negative scenario. The MSC precondition is made of the interaction *start*; the prohibited event is *open doors*. In our tool, prohibited events in

negative MSCs appear below a (red) dashed line. The negative scenario in Fig.1.b is used to express that the train controller may not open the doors after having started the train (without any intermediate interaction). Fig.1.c and 1.d show other positive scenarios.

2.2 Behavior models as Labelled Transition Systems

A system is behaviorally modeled as a set of concurrent state machines – one per agent. Each agent is characterized by a set of states and a set of transitions between states. Each transition is labelled by an event. Our state machines are labelled transition systems (LTS) [17].

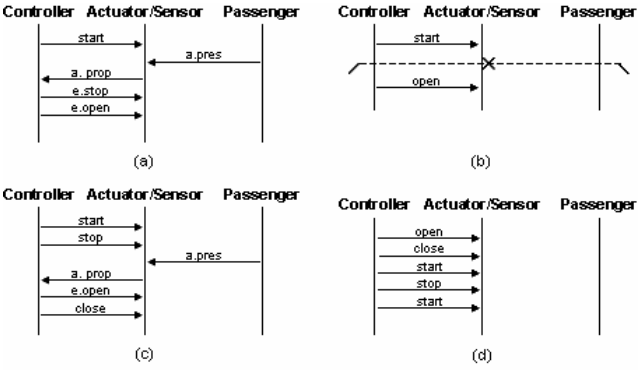

Figure 1 - Scenarios for a train system

A LTS is an automaton defined by a structure (Q,Σ,δ,q_0) where Q is a finite set of states, Σ is a set of event labels, δ is a transition function mapping $Q\text{x}\Sigma$ to 2^Q, and q_0 is the initial state.

A complex system is modelled by parallel composition of LTS models of its components. The composed models behave asynchronously but synchronize on shared events.

The semantics of MSCs is defined in terms of LTS and parallel composition [22]. A MSC timeline defines a unique finite LTS execution that captures a corresponding agent behavior. Similarly, the semantics of an entire MSC is defined in terms of the LTS modeling the entire system. MSCs define executions of the system LTS, that is, the parallel composition of each agent LTS.

2.3 Goals as fluent-based assertions

A *goal* is a prescriptive statement of intent whose satisfaction requires the cooperation of agents forming the system. Unlike goals, *domain properties* are descriptive statements about the environment – such as physical laws, organizational rules, etc. Goals are structured into AND/OR refinement graphs showing how they contribute to each other [11].

In this paper, we will formalize goals and domain properties in Fluent Linear Temporal Logic (FLTL) [3]. This formalism is convenient for specifying temporal logic properties over an event-based operational model.

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. A fluent definition takes the form:

fluent Fl = < $Init_{Fl}$, $Term_{Fl}$ > initially $Initially_{Fl}$

In our train example, the fluents *DoorsClosed* and *Moving* are defined as follows:

fluent DoorsClosed = <{*close doors*},
{*open doors*, *emergency open*}>
initially ***true***

fluent Moving = <{*start*},
{*stop*, *emergency stop*}>
initially ***false***

A fluent *Fl* holds at some time if either of the following conditions holds:

(a) *Fl* holds initially and no terminating event has yet occurred;

(b) some initiating event has occurred and no terminating event has occurred since then.

A fluent is *controlled* by an agent if the agent controls all initiating and terminating events of the fluent. It is monitored by an agent if the agent controls or monitors all initiating and terminating events of the fluent.

The FLTL assertions for goals and domain properties use standard operators for temporal referencing such as: $\mathbf{o}$ (at the next smallest time unit), $\Diamond$ (some time in the future), $\Box$ (always in the future), $\mathcal{U}$ (always in the future until), $\mathcal{W}$ (always in the future unless), $\rightarrow$ (implies in the current state), $\Rightarrow$ (always implies), see [18, 10]. For example, the goal "the doors shall remain closed while the train is moving" can be formalized in terms of the above fluents as follows:

DoorsClosedWhileMoving = $\Box$ (Moving $\rightarrow$ DoorsClosed)

2.4 LTS synthesis as grammar induction

Our LTS synthesizer proceeds in two steps [1]. First, the input scenarios are generalized into a LTS for the entire system, called *system LTS*. This LTS is then projected on each agent using standard automaton transformation algorithms [5].

The system LTS covers all positive scenarios and excludes all negative ones. It is obtained by an interactive extension of a grammar induction algorithm known as RPNI [20]. Grammar induction aims at learning a language from a set of positive and negative strings defined on a specific alphabet. The alphabet here is the set of event labels; the strings are provided by positive and negative scenarios.

RPNI first computes an initial LTS solution, called *Prefix Tree Acceptor* (PTA). The PTA is a deterministic LTS built from the input scenarios; each scenario is a branch in the tree that ends with a "white" state, for a positive scenario, or a "black" state, for a negative one. As in the other synthesis approaches mentioned in Section 1, scenarios are assumed to start in the same system state.

Fig.2 shows the PTA computed from the scenarios in Fig.1. A black state is an error state for the system. A path leading to a black state is said to be *rejected* by the LTS; a path leading to a white state is said to be *accepted* by the LTS. By construction, the PTA accepts all positive input scenarios while rejecting all negative ones.

Behavior generalization from the PTA is achieved by a "generate-and-test" algorithm. This algorithm performs an exhaustive search for equivalent state pairs that are *merged* into equivalence classes. Two states are considered *equivalent* if they have *no incompatible continuation*, that is, there is no subsequent event sequence accepted by one and rejected by the other.

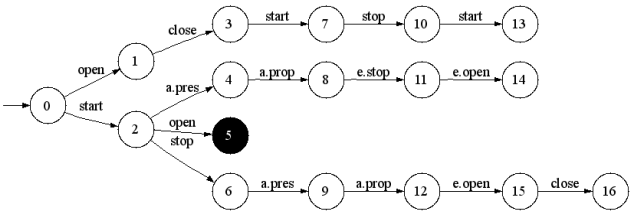

Figure 2 - PTA built from the scenarios in Fig. 1

At each generate-and-test cycle, RPNI considers merging a state q in the current solution with a state q' of lower rank. Merging a state pair (q,q') may require further merging of subsequent state pairs to obtain a deterministic solution; shared continuations of q and q' are folded up by such further merges. When this would end up in merging black and white states, the merging of (q,q') is discarded, and RPNI continues with the next candidate pair.

Fig.3 shows the system LTS computed by the tool for our train example, with the following partition into equivalence classes:

$$\pi = \{\{0,3,6,10,16\}, \{1,14,15\}, \{2,7,13\}, \{4\}, \{5\}, \{8\}, \{9\}, \{11,12\}\}$$

The equivalence relation used by this inductive algorithm shows the important role played by negative scenarios to avoid merging non-equivalent system states and derive correct generalizations. RPNI is guaranteed to find the correct system LTS when the input sample is rich enough [20]; two distinct system states must be distinguished in the PTA by at least one continuation accepted from one and rejected from the other. When the input sample has no enough negative scenarios, RPNI tends to compute a poor generalization by merging non-equivalent system states.

To overcome this problem, our synthesizer extends RPNI in two directions:

- *Blue Fringe search:* The search is made heuristic through an evaluation function that favors states sharing common continuations as first candidates for merging [13].
- *Interactive search:* The synthesis process is made interactive through scenario questions asked by the synthesizer whenever a merged state gets new outgoing transitions [1].

To answer a scenario question, the user has just to accept or reject the new MSC scenario shown by the synthesizer. The answer results in confirming or discarding the current candidate state merge. Scenario questions provide a natural way of eliciting further positive and negative scenarios to enrich the scenario sample.

Fig.4 shows a scenario question that can be rephrased as follows: "if the train starts and a passenger presses the alarm button, may the controller then open the doors in emergency and close the doors afterwards?". This scenario should be rejected as the train may not move with open doors.

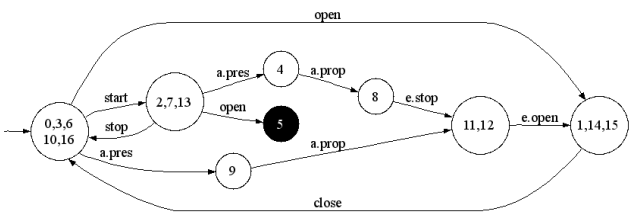

Figure 3 - System LTS for the train example

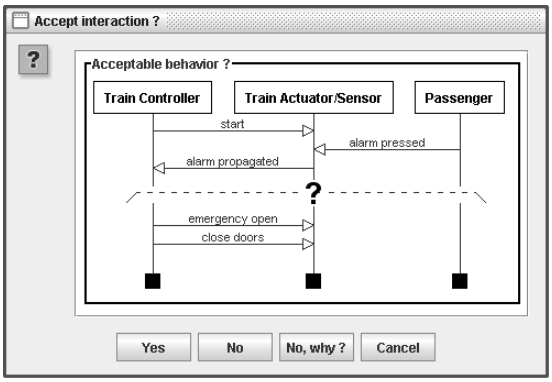

Figure 4 - Scenario question generated during synthesis

3 OPTIMIZING THE SYNTHESIS OF BEHAVIOR MODELS

The synthesizer outlined in the preceding section shares a number of problems with other inductive approaches to model synthesis. Unlike deductive inference, induction is not necessarily sound; overgeneralization can occur. The technique is also sensitive to classification errors. Moreover, the synthesized model may not be consistent with known properties of the domain and/or with known goals of the target system. Specifically to our approach, the number of scenario questions might become too large for interaction-intensive systems.

This section discusses further techniques we have implemented in our synthesizer, called ISIS, to address those problems. (ISIS stands for *Interactive State machine Induction from Scenarios*.)

Section 3.1 shows how the induction process can be constrained through domain knowledge. The integration of goals is then discussed in Section 3.2. As we will see there, a temporal logic specification embodies many negative scenarios; such integration provides a drastic remedy to the potential lack of enough negative scenarios.

The optimization techniques detailed hereafter are based on various equivalence relations on system states beside the RPNI one which focuses on compatibility of continuations. We use the term *equivalence relation* here in its usual mathematical sense, that is, a symmetric, reflexive, and transitive binary relation over states. The general principle underlying our techniques is the following:

> *Two states will be considered for merging if they agree according to all considered equivalence relations.*

The equivalence relations considered hereafter are all invariant under state merging; a state derived by merging some states simply inherits their relation. This allows each relation to be computed *only once* on the initial PTA. The results of such pre-processing are kept as annotations on PTA states.

Note that the above principle for state merging is very general. It could therefore be further instantiated to other equivalence relations not considered in this paper.

3.1 Injecting domain knowledge in the synthesis process

The domain knowledge used to constrain state merging comes from multiple sources: fluent definitions, knowledge about components in the environment of the software-to-be, and FLTL specifications of domain properties. We discuss these successively.

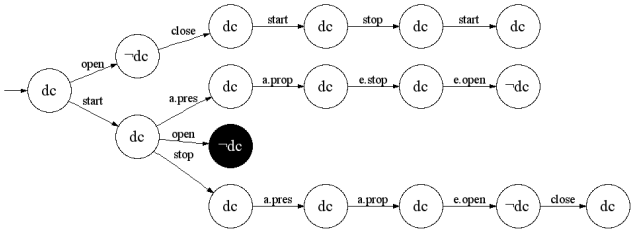

Figure 5 – Propagating fluent values along a PTA
(*dc* is a shorthand for ***DoorsClosed***)

3.1.1. Propagating fluents

Fluent definitions provide simple and natural domain descriptions to constrain induction. For example, the definition

fluent DoorsClosed = < {close doors},
{open doors, emergency open} >
initially **true**

describes train door states as being either closed (DoorsClosed = true) or open (DoorsClosed = false), and describes which event is responsible for which state change.

Such descriptions can be effectively used to constrain the induction process so that the synthesized system LTS conforms to them. The idea is to compute the value of every fluent at each PTA state by symbolic execution; the PTA states are then decorated with the conjunction of such values. The pruning rule for constraining the induction process is here to *avoid merging inconsistent states*, that is, to avoid merging two states whose decoration has at least one fluent with different values.

The specific equivalence relation here is thus the set of state pairs where both states have the same value for every fluent. *Two states will be considered for merging if they have the same value for every fluent.*

The decoration of the merged state is simply inherited from the states being merged.

To compute PTA node decorations by symbolic execution, we use a simplified version of an algorithm described in [1] to propagate fluent definitions forwards along paths of the PTA tree. (The general algorithm propagates fluent definitions along multiple paths with cycles until a fixpoint is reached.)

Fig.5 shows the result of propagating the values of fluent DoorsClosed, according to its definition in Section 2.3, along the PTA shown in Fig. 2.

3.1.2. Unfolding models of external components

Quite often the components being modelled need to interact with external components in their environment – e.g., legacy components in a bigger existing system, foreign components in an open system, and so forth. In such cases the behavior of external components is generally known – typically, through some behavioral model [4]. Here we assume that external components are known by their LTS model.

For example, Fig.6 shows the LTS for a legacy alarm sensor in our train system. When the alarm button is pressed by a passenger, this component propagates a corresponding signal to the train controller.

A LTS model of an external component can constrain the induction

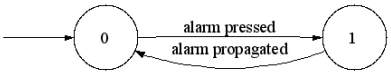

Figure 6 – LTS model for an alarm sensor

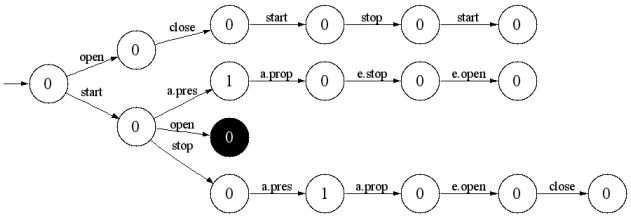

Figure 7 – Unfolding the alarm sensor LTS on the PTA

process so that the synthesized system LTS conforms to it. The idea is to decorate the PTA with states of the external LTS by unfolding the latter on the PTA. Such decoration is performed by jointly visiting the PTA and the external LTS; the latter synchronizes on shared events and stays in its current state on other events.

Fig.7 shows the result of unfolding the alarm sensor LTS from Fig. 6 on the PTA shown in Fig. 2. Each state in Fig. 7 is labelled with the number of the corresponding state in the alarm sensor LTS.

The pruning rule for constraining the induction process is now to *avoid merging states decorated with distinct states of the external component.* The specific equivalence relation used here is the set of states where both states have the same external LTS state. *Two states will be considered for merging if they have the same external LTS state.*

3.1.3. Using declarative domain properties

Descriptive statements and assumptions about the domain can be expressed declaratively in FLTL. For example, the physical law

$$\Box\,(\text{HighSpeed} \rightarrow \text{Moving})$$

excludes all negative scenarios where the train is running at high speed while not moving.

The technique for constraining induction through descriptive or prescriptive statements is the same; we discuss it hereafter.

3.2 Injecting goals in the synthesis process

Goals are prescriptive statements of intent about the target system. We restrict our attention here to goals that can be formalized as FLTL safety properties. For such goals we can use the technique described in [3] to generate a tester. A *tester* for a property is a LTS extended with an error state such that every path leading to the error state violates that property.

Consider the goal requiring train doors to remain closed while the train is moving:

$$\text{DoorsClosedWhileMoving} = \Box\,(\text{Moving} \rightarrow \text{DoorsClosed})$$

Fig.8 shows the tester LTS for this property (the error state is the black one). Any event sequence leading to the error state from the initial state corresponds to an undesired system behavior. In particular, the event sequence <*start*, *open*> corresponds to the

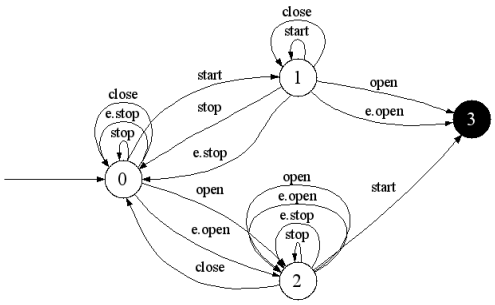

Figure 8 - Tester LTS for the goal *DoorsClosedWhileMoving*

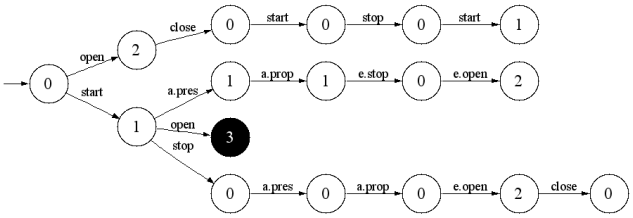

Figure 9 – PTA decorated using the tester LTS from Fig. 8

initial negative scenario in our running example (see Fig.1). As seen in Fig.8, the tester provides many more negative scenarios. Property testers can in fact provide potentially infinite classes of negative scenarios.

The property tester is used to constrain the induction process in a way similar to an external component LTS. The PTA and the tester are traversed jointly in order to decorate each PTA state with the corresponding tester state. Fig. 9 shows the PTA decorated using the tester in Fig.8.

The pruning rule for constraining the induction process is now to *avoid merging states decorated with distinct states of the property tester.* The specific equivalence relation used here is the set of states where both states correspond to the same property tester state. *Two states will be considered for merging if they have the same property tester state.*

This pruning technique has the additional benefit of ensuring that the synthesized system LTS satisfies the considered goal or domain property. A proof argument is provided by known results from automata theory. A tester for a safety property is a canonical automaton, that is, minimal and deterministic [3]. A bijection thus exists between states and continuations [5]. In other words, two states are distinct if and only if there is at least one continuation to distinguish them. In the particular case of the tester LTS, two states are distinct if and only if they do not have the same set of continuations leading to the error state (these are called *violating continuations*).

The equivalence relation and corresponding pruning rule thus amount to avoid merging system states that do not share the same set of violating continuations.

In practice, ISIS reuses the LTL2Buchi tool [2] and generates property testers from the produced Büchi automata.

4 MINING GOALS FROM SCENARIOS

At this point the question arises as to where those goals and domain descriptions are coming from. In view of our emphasis on end-user involvement, possibly with an analyst on the back, we would like to obtain them systematically through property mining from scenarios. This section discusses how goals and domain descriptions can be obtained as explanations of *why* a scenario is rejected as counterexample (Section 4.1), or as safety properties inferred automatically from positive scenarios (Section 4.2).

Scenarios are fragmentary, operational, and leave the underlying goals, assumptions, and domain properties implicit. Mining these provides further benefits beside this paper's concern of reducing the number of scenario questions and synthesizing models that are "correct" with respect to them. Goal specifications can be used to formally check or derive goal refinements, generate hazard and threat conditions, detect conflicts, and generate further scenarios that satisfy them [11, 12].

4.1 WHY questions about negative scenarios

When the user answers a scenario question by rejecting the scenario as counterexample, the ISIS synthesizer asks her the reason for rejection. The user can explain it by adding the goal or domain description being violated by the scenario.

For example, when rejecting the scenario question in Fig.4, the user may click the "No, why ?" button in order to explain why that behavior is prohibited. The lower part of the window in Fig.10 shows the reason for rejection; the goal

$$\square\,(\text{Moving} \rightarrow \text{DoorsClosed})$$

has been added using the fluents defined in the middle part of the window.

No scenario question excluded by the goal will be asked subsequently in the induction process. Moreover, any inconsistency between the added property and the scenarios previously entered is automatically detected and reported.

4.2 Inferring goals from scenarios

A more ambitious objective is to infer safety properties automatically from the scenario collection. We focus on specific property patterns, under responsibility of single agents. An inferred property will therefore be a requirement on a software agent, an assumption on an environment agent, or a domain property [11].

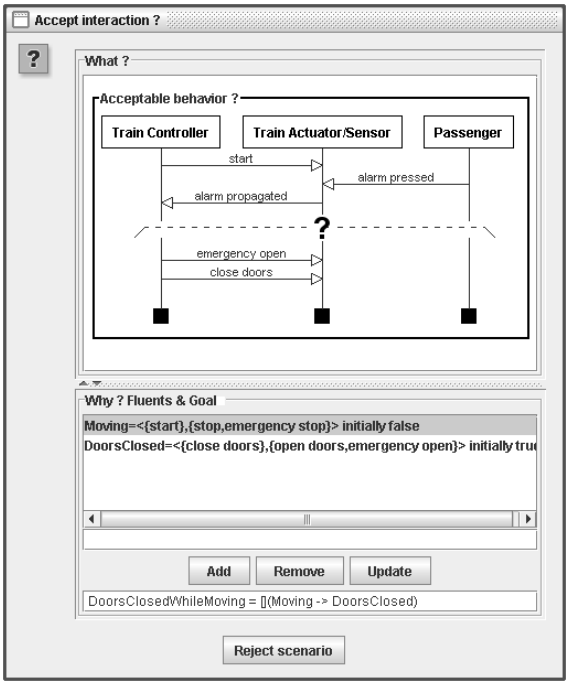

Figure 10 – Negative scenario explained by the goal *DoorsClosedWhileMoving*

Our procedure for inferring property specifications from scenarios is specified as follows:

GIVEN a set of positive scenarios,
a set of fluents,

FIND a conjunctive set of properties covering all input scenarios, and taking the form:

- $\square\,(A \rightarrow C)$ (*Maintain/Avoid* goals)
- $\square\,(A \rightarrow \circ C)$ (*Immediate Response* goals)

where *A* and *C* are state assertions without temporal operators.

In previous work, we have developed an inductive learning technique for generating LTL goal specifications from positive and negative scenarios [10]. This technique is not applied here because it is not based on fluents, and makes the stronger assumption that specifications of interaction events are available as pre- and postconditions on corresponding operations.

The inference of *Maintain/Avoid* properties is detailed in Section 4.2.1 while the inference of *Immediate Response* properties is discussed in Section 4.2.2.

4.2.1. Inferring Maintain/Avoid properties

Let us consider the two positive scenarios in Fig.11. The state predicates *S0*, *S1*, *S2*, *S3*, *S4*, *S5* along agent *B*'s timelines are fluent conjunctions that hold between the corresponding interactions. The following assertion on agent *B* then holds in any of *those* states:

$$S0 \vee S1 \vee S2 \vee S3 \vee S4 \vee S5$$

Let *Inv* denote this assertion. If those two scenarios were to cover all states of agent *B*'s LTS, we could infer that *Inv* is preserved under all behaviors of this agent and get the safety property:

$$\Box\ Inv$$

As the set of positive scenarios in which the agent is involved is likely not to be complete, this generalization can be unsound; it must therefore be validated by the user.

It is thus crucial for the candidate property to be structured and easy to understand. To achieve this we transform it into a minimal conjunctive normal form (CNF). This yields a logically equivalent set of simpler and shorter candidate invariants.

Rejection of one of these by the user means that the scenario collection is not complete. ISIS then asks the user to provide a counterexample to the rejected invariant to be covered as additional positive scenario.

The procedure for inferring *Maintain/Avoid* properties on a single agent is now detailed step by step on our running example. We start from the positive scenarios in Fig.1 together with the following fluent definitions:

fluent *DoorsClosed* = < {*close doors*},
{*open doors*, *emergency open*} >
initially **true**

fluent *Moving* = < {*start*},
{*stop*, *emergency stop*} >
initially **false**

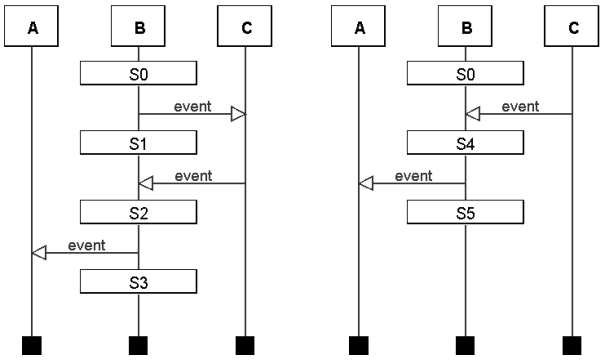

Figure 11 - State predicates along timelines of an agent in multiple scenarios

fluent *Alarmed* = < {*alarm propagated*},
{*emergency open*} >
initially **false**

Step 1: Compute fluent-based state predicates along the agent's timelines

To obtain the local state predicates S_i along the agent*'s* timelines, we decorate these timelines with the fluents holding at the corresponding point.

As we are looking for prescriptions (or descriptions) on this agent, we restrict ourselves to the fluents *controlled* by the agent. Properties of the form $\Box\ (A \rightarrow B)$ where *A* contains monitored fluents and *B* contains controlled ones would be unrealizable by the agent [14]; we are not making the synchrony hypothesis where agents can react instantly to their environment. Said otherwise, if *A* becomes *true*, the agent needs at least one (smallest) time unit to set *B* to *true*.

Conjunctions of controlled fluents that hold at specific points of an agent timeline are called *controlled predicates*. The algorithm for decorating timelines with controlled predicates is another simplified version of our symbolic execution algorithm [1], particularized here to a MSC timeline instead of an entire LTS. This algorithm is a fluent-based counterpart of the algorithm for generating condition lists along timelines in [10], also used in [23].

Let us focus on the TrainController agent. It controls the fluents DoorsClosed and Moving because it performs their initiating and terminating events. The decoration is generated from top to bottom. First, we annotate the initial state *S0* of the timeline with the initial fluent values, yielding $\neg$ Moving $\wedge$ DoorsClosed as Moving is initially *false* and DoorsClosed is initially *true*. Recursively, we compute the next state decoration using the above definitions of fluents *Moving* and *DoorsClosed*. After the *start* event, the decoration is *Moving* $\wedge$ *DoorsClosed* because *start* is an initiating event of *Moving*; the fluent *DoorsClosed* does not change as *start* is not among its initiating/terminating events. We continue until the end of the scenario is reached (see Fig.12).

Step 2: Form the candidate invariant

A single scenario contributes to the candidate global invariant through the following assertion:

$$InvSc = \bigvee_{i \in 0...n} cp_i$$

where cp_i is the controlled predicate decorating the timeline right after event *i*. In our example we get (see Fig.12):

$$\neg \text{Moving} \wedge \text{DoorsClosed} \vee \text{Moving} \wedge \text{DoorsClosed}$$
$$\vee \text{Moving} \wedge \text{DoorsClosed} \vee \text{Moving} \wedge \text{DoorsClosed}$$
$$\vee \neg \text{Moving} \wedge \text{DoorsClosed} \vee \neg \text{Moving} \wedge \neg \text{DoorsClosed}$$

Similarly we compute the contributions of all other input scenarios to obtain the candidate global invariant:

$$Inv = \bigvee_j InvSc_j$$

Step 3: Normalize the candidate invariant in minimal CNF form

The candidate global assertion is then transformed into a minimal conjunction of disjunctions, using standard CNF tools. In our example, *Inv* is reduced to

$$\neg \text{Moving} \vee \text{DoorsClosed}.$$

Step 4: Generalize the candidate CNF invariant

Generalization to any state is simply achieved by prefixing *Inv* with the "always" operator. In our example, we get

$$\Box\ (\neg \text{Moving} \vee \text{DoorsClosed}).$$

Step 5: Validate each conjoined property with the user

Each conjunct in the generalized CNF invariant is shown to the user for validation. There is a readability issue here as each conjunct can be presented in alternative, equivalent ways. In our example, we obtain one requirement only that can be presented in one of the following forms:

$\Box$ ($\neg$ Moving $\vee$ DoorsClosed)

$\Box$ (Moving $\rightarrow$ DoorsClosed)

$\Box$ ($\neg$ DoorsClosed $\rightarrow$ $\neg$ Moving)

$\Box$ $\neg$ ($\neg$ DoorsClosed $\wedge$ Moving)

Heuristics for increased readability are needed here. So far we favored implications with minimal number of negation symbols – but other heuristics should deserve further attention.

The user may react in three ways with respect to each candidate property in the conjunctive list presented.

- *Accept it:* The property is added to the specification and used to constrain the induction process (see Section 3.2).
- *Reject it:* This means that the current scenario collection does not cover all agent states. The user is then asked to provide a counterexample to the rejected property in order to enrich the collection with a new positive scenario to be covered. Property inference thus turns to be an effective way of checking whether a scenario collection is complete enough and, if not, solicit new behavior examples.
- *Don't know:* the generated property might be hard to understand. In this case, it may be ignored.

4.2.2. Inferring Immediate Response Properties

The second property pattern that ISIS can infer takes the form

$$\Box (A \rightarrow \mathbf{o}\, B)$$

where *A* may contain monitored fluents and *B* contains controlled fluents only. This stimulus-response property pattern is important as it arises fairly frequently in reactive, event-based systems.

Such properties are not necessarily closed under stuttering [9]; their satisfaction by an event trace may be affected by insertion or removal of unobservable events. In the *Immediate Response* properties we consider, the "next" operator means "in the next smallest time unit" and refers to the alphabet of the input scenarios. We are protected against stuttering as long as such properties are not combined with others defined on another alphabet. The inferred *Immediate Response* properties must thus be handled with care – e.g., they should not be used when composing the system with components on different alphabets. In any case, they need to be validated by the user as well.

The inference of *Immediate Response* properties is fairly similar to *Maintain/Avoid* properties. We briefly discuss the differences.

Step 1: Compute fluent-based state predicates along the agent's timelines. Each agent's timeline is decorated with its controlled *and monitored* fluents by use of the same algorithm. Conjunctions of controlled and monitored fluents that hold at specific points of an agent timeline are called *state predicates*. For our TrainController agent, the timelines are decorated using three fluents: the controlled fluents *Moving* and *DoorsClosed*, and the monitored fluent *Alarmed*.

Step 2: Form the candidate invariant. The single-scenario invariant takes a different form here, namely,

$$\text{InvSc} = \bigvee_{i \in 0\ldots n-1} (sp_i \wedge \mathrm{o}\, cp_{i+1})$$

where sp_i is the state predicate right after event *i* and cp_i the controlled predicate after event *i*.

The global candidate invariant on all scenarios is then:

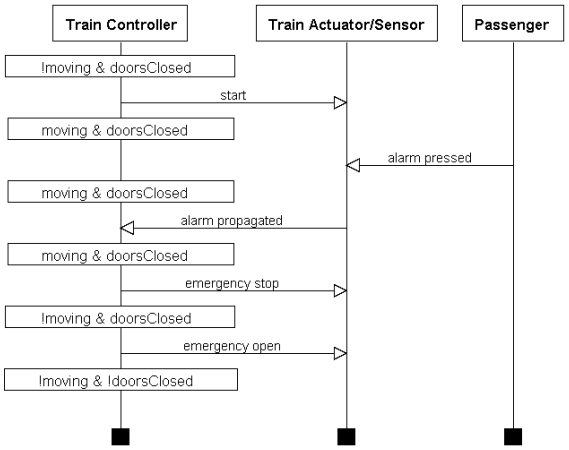

Figure 12 - Train controller decorated with controlled fluents

$$Inv = \bigvee_j \text{InvSc}_j$$

Step 3: Normalize the candidate invariant in minimal CNF form. This step is similar. For the three positive scenarios in Fig. 1 we obtain:

Inv = ($\neg$ Alarmed $\vee$ o $\neg$ Moving)
$\wedge$ (DoorsClosed $\vee$ $\neg$ Moving)
$\wedge$ (DoorsClosed $\vee$ o $\neg$ Moving)
$\wedge$ ($\neg$ Moving $\vee$ o DoorsClosed)
$\wedge$ (o DoorsClosed $\vee$ o $\neg$ Moving)
$\wedge$ (DoorsClosed $\vee$ o DoorsClosed)
$\wedge$ ($\neg$ Alarmed $\vee$ Moving $\vee$ o $\neg$ DoorsClosed)

Such disjunctions can be rewritten as implications taking the form

$$A \rightarrow \mathrm{o}\, B$$

where *A* is a conjunction and *B* a disjunction.

Step 4: Generalize the candidate CNF invariant. This step is similar; we just we add the "$\Box$" prefix.

Step 5: Validate candidate properties with the user. Here we perform some filtering on the properties shown to the user.

- Properties without "next" operator are not presented as they are already shown as *Maintain/Avoid* properties.
- Properties where all fluents are prefixed with the "next" operator are not presented for a similar reason, e.g., $\Box$(o DoorsClosed $\vee$ o $\neg$ Moving)
- Tautologies resulting from the LTS semantics are not shown.

Let us explain the last point. Consider the following fluents:

Fluent A = $\langle a_{Init}, a_{Term} \rangle$

Fluent B = $\langle b_{Init}, b_{Term} \rangle$

together with the assertion

$$\Box (A \wedge B \rightarrow \mathrm{o}\, A \vee \mathrm{o}\, B)$$

This assertion is a tautology in view of the LTS *one-input assumption*. This semantic assumption states that exactly one input event occurs at every state transition. As a_{Term} and b_{Term} cannot occur at the same time, if *A* and *B* are true in the current state, either *A* or *B* will be true in the next state.

A tautology tester is a universal LTS, that is, a LTS accepting any sequence of events. The error state for such testers is unreachable. No scenario can violate the property. Tautologies thus do not constrain the induction process. ISIS generates testers for all

Specification	Events	States	Edges	SC+	SC-
Mine Pump	8	10	13	3	0
Big Train	13	17	23	3	0
Phone System	16	23	33	6	3

Table 1 – Size of case studies

inferred properties in the candidate CNF invariant, and presents only properties whose tester is not the universal LTS.

In our example, there is no tautology. Five generated properties are shown to the user, e.g., the requirement

$$\Box\,(\text{Alarmed} \rightarrow \text{o} \neg \text{Moving})$$

which states that "*when the train is in alarm state, the train controller must stop the train immediately*".

Here again, the user is asked to provide a counterexample scenario when a property is rejected. All accepted properties are added to the specification to constrain the induction process.

5 EVALUATION

We compared how the techniques in Sections 3 and 4 perform on three different case studies of varying complexity. The first case study is a mine pump system inspired from [7]. The second is an extended, less simplistic version of the train system used here as a running example. The third is a phone system handling communications between a caller and a callee.

The objective was to assess the impact of constraining induction through fluents, models of external components, domain descriptions, and goals. Impact was measured in terms of number of generated scenario questions and adequacy of the synthesized models.

For each case study, we proceeded in two steps:

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

In Step 1, condition (a) amounts to require the scenario collection to be *structurally complete* [20, 1]; every transition in the LTS being synthesized must occur in at least one scenario. We used the ISIS tool itself to incrementally set up such a scenario collection. We started from an initial set of scenarios that end-users would typically provide. By generating scenario questions, validating inferred properties, and validating synthesized LTSs, we found a number of additional scenarios that were missing for the scenario collection to be usable for the comparisons in Step 2.

The size of the scenario collection resulting from Step 1 is shown in Table 1. "SC+" and "SC-" correspond to the number of positive and negative scenarios, respectively. This table also shows the size of the target LTS in terms of number of different event labels, states, and edges. The average size of scenarios, in terms of number of interactions, is not shown there. This size is *8* for the Mine Pump system, *9* for the Big Train system, and *11* for the Phone System.

To perform the comparisons, an oracle was implemented to simulate the end-user. This oracle knows the target LTS for each specification and correctly classifies scenario questions as positive or negative.

Specification	Algorithm	Q+	Q-	Model Accuracy
Mine Pump	RPNI	1	30	missing/unallowed paths
	BlueFringe	1	4	adequate model
Big Train	RPNI	4	83	adequate model
	BlueFringe	5	5	adequate model
Phone System	RPNI	5	171	missing/unallowed paths
	BlueFringe	5	19	missing/unallowed paths

Table 2 – RPNI vs. BlueFringe

Comparison 1: RPNI vs. Blue-Fringe

Table 2 shows the number of questions the oracle had to answer and the adequacy of the generated LTS, in the three case studies, when no additional knowledge is used to constrain induction. A synthesized model is said to be *adequate* if it matches the target known LTS. "Q+" and "Q-" are the number of accepted and rejected scenario questions, respectively.

Note that the number of rejected scenario questions is drastically reduced thanks to Blue-Fringe's heuristic search. For bigger systems pure RPNI becomes unusable. In the Phone system, an adequate LTS cannot be synthesized from the scenario collection. Wrong generalizations do occur; some states are merged whereas they need to be distinguished in the adequate model. Finally, the number of rejected scenarios tends to be much larger than the number of accepted ones. This observation confirms the usefulness of scenario questions; negative answers force the induction algorithm to backtrack when an incorrect search path has been taken.

As Blue-Fringe is seen to be by far superior to pure RPNI, we will keep Blue-Fringe only for further comparisons.

Comparison 2: Impact of fluent propagation

Table 3 shows the influence of fluent decorations to constrain the induction process. Note that the number of rejected scenario questions is decreasing in each case study as the number of fluents is increasing. Such questions can even disappear when the set of fluent definitions is sufficient. For the same generated LTS, the number of accepted scenario questions remains the same; fluent-based state information only allows state merges to be rejected. Also note that two fluent definitions in the Phone system are sufficient for an adequate model to be found.

Comparison 3: Impact of inferred properties

From the fluent definitions and the same initial scenario collection, ISIS automatically inferred various important requirements and

Specification	Nb fluents	Q+	Q-	Model Accuracy
Mine Pump	0	1	4	adequate model
	1	1	1	adequate model
	2	1	0	adequate model
	3	1	0	adequate model
Big Train	0	5	5	adequate model
	1	5	3	adequate model
	2	5	3	adequate model
	3	5	3	adequate model
	4	5	2	adequate model
	5	5	0	adequate model
Phone System	0	5	19	missing/unallowed paths
	1	5	13	missing/unallowed paths
	2	6	9	adequate model
	3	6	4	adequate model

Table 3 – Impact of fluent propagation on induction

Specification	# properties	Q+	Q-	Model Accuracy
Mine Pump	0	1	4	adequate model
	1	1	0	adequate model
	2	1	0	adequate model
	3	1	0	adequate model
Big Train	0	5	5	adequate model
	1	5	3	adequate model
	2	5	3	adequate model
	3	5	2	adequate model
	4	5	2	adequate model
	5	5	2	adequate model
Phone System	0	5	19	missing/unallowed paths
	1	6	6	adequate model
	2	6	4	adequate model
	3	6	4	adequate model

Table 4 – Impact of inferred properties on induction

domain properties. As the scenario collection covers every event label of the synthesized LTS (see the structural completeness condition (a) above), all inferred properties are adequate.

For the *Mine Pump* system, the three main requirements were inferred automatically, e.g.,

> *When the water level is below the low water threshold, the pump controller must immediately set the pump to "off".*

For the *Big Train* system, three requirements and two domain properties were inferred automatically, e.g.,

> *The train may never run at high speed when it comes near a station.*

For the *Phone* system, three requirements were inferred automatically, e.g.,

> *When the caller hangs up, the connection should immediately be closed.*

The inferred properties were used in turn to incrementally constrain the induction process. Table 4 shows the results. The same observations can be made as with fluents. However, goals and domain properties are seen to be more powerful than fluents. With one single goal, there are no rejected scenario questions anymore in the Mine Pump system; the LTS generated for the Phone system is now adequate. (The fact that the numbers of fluents and goals are the same in these case studies is purely coincidental.)

Comparison 4: Combined use of fluents, properties, and models of external components

Table 5 shows the results of a Blue-Fringe induction constrained with available fluents, goals, and domain properties, plus one external component in each case study. Comparing this table with Table 2 shows how much is gained when the various techniques described in this paper are combined to constrain the interactive LTS synthesis process.

6. CONCLUSION

Model-driven development requires adequate models to be developed. Model building is a complex task, especially in the case of behavior models. Techniques and tools are therefore needed to support this task. System stakeholders should ideally be involved, at least in the first stages of the process.

Specification	Q+	Q-	Model Accuracy
Mine Pump	1	0	adequate model
Big Train	5	0	adequate model
Phone System	7	2	adequate model

Table 5 – Combining fluents, properties, and external components to constrain induction

This paper has presented a number of techniques that provide significant improvements over a previous method for synthesizing behavior models interactively from end-user scenarios [1]. These improvements are based on the use of knowledge about the target system to constrain the induction process. Such knowledge includes fluent definitions, behavior models of external components, domain properties, and system goals. Each type of knowledge allows a specific equivalence relation to be defined for pruning the search space of mergeable state pairs. As a result, the number of scenario questions in end-user interactions is reduced significantly, sometimes drastically; and the adequacy of the synthesized model is improved. Some of the techniques may require analyst intervention to formalize specifications or to explain such specifications in natural language. To address this limitation, an important class of safety properties is inferred automatically from scenarios. All techniques presented here are implemented in the ISIS synthesizer. The evaluation of this tool on case studies of growing complexity appears encouraging. We are currently pursuing such evaluation further on real web applications where the number of states and transitions is significantly larger.

An additional strength of our inductive approach is to allow system knowledge to be integrated quite easily for constraining induction. Two states are considered for merging if and only if they agree according to all considered equivalence relations. Further improvements with new specific equivalence relations on states could be incorporated at low cost. For further pruning we might also propagate non-equivalence backwards, from non-equivalent states, along PTA transitions with the same event label.

Declarative properties were seen to play a prominent role in effective pruning of the inductive search space. Every property embodies a whole class of positive scenarios and rules out many negative scenarios that will not be generated as questions. Properties contribute to structural completeness of the scenario sample much more effectively than isolated scenarios. Unlike model checking approaches, such properties are not "floating in the air"; they pertain to structured goal models as system goal, software requirement, environment assumption, or domain property [11]. Some of them can be taken from available goal models, others can be inferred automatically and integrated into such models.

The current version of ISIS raises a number of open issues. As in all other model synthesis approaches, the input scenarios to be initially provided are assumed to start in the same state. This assumption appears too strong for, e.g., scenario fragments coming from multiple end-users. Our approach is also highly sensitive to classification errors by end-users. The consistency checks performed when a property is entered is a first step to address this.

The effectiveness of our techniques may depend on the choice of fluents. What makes a good choice of fluents and how to identify these is another interesting issue to consider.

The state machines ISIS generates are "flat" LTS. We plan to exploit fluent-based information to parallelize them. The projection of the synthesized LTS on local agents is known to possibly introduce additional behaviors [1]. Such behaviors, called implied scenarios, can be detected using techniques described in [21, 15]. Techniques to eliminate undesirable implied scenarios still need to be developed.

In any case, the more we work on the model synthesis problem, the more we are convinced that the "goal – scenario – state machine" triangle is a rich, not yet fully exploited, source of synergy and mutual reinforcement for model analysis and synthesis.

ACKNOWLEDGMENTS

This work was partially supported by the Regional Government of Wallonia (ReQuest project, RW Conv. 315592). Warmest thanks are due to Pierre Dupont. Beyond helpful comments on a previous version of this paper, Pierre pointed out that the LTS synthesis problem can be seen as a grammar induction problem, and observed that the various techniques presented in this paper are all based on specific equivalence relations on states.

REFERENCES

[1] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde, "Generating Annotated Behavior Models From End-User Scenarios", *IEEE Trans. on Software Engineering,* Special Issue on Interaction and State-Based Modeling*,* Vol. 31 No.12, Dec. 2005, 1056-1073.

[2] D. Giannakopoulou and F. Lerda, *LTL2Buchi*, available at http://ic.arc.nasa.gov/people/dimitra/LTL2Buchi.php.

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

[4] R.J. Hall and A. Zisman, "OMML: A Behavioral Model Interchange Format", *Proc. RE'04, 12th IEEE Joint International Requirements Engineering Conference*, Kyoto, Sept. 2004.

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

[6] M.Jarke and R. Kurki-Suonio (eds.), Special Issue on Scenario Management, *IEEE Trans. on Sofware. Engineering*, Vol. 24 No. 12, Dec. 1998.

[7] M. Joseph. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall Intl., 1996.

[8] I. Kruger, R. Grosu, P. Scholz and M. Broy, From MSCs to Statecharts, *Proc. IFIP WG10.3/WG10.5 Intl. Workshop on Distributed and Parallel Embedded Systems* (Scholoß Eringerfeld, Germany), F. J. Rammig (ed.), Kluwer, 1998, 61-71.

[9] L. Lamport, "The Temporal Logic of Actions", *ACM Transactions on Programming Languages and Systems*, 16(3), 1994, 872-923.

[10] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Sofware. Engineering*, Vol. 24 No. 12, December 1998.

[11] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective". Keynote Paper, *Proc. ICSE'2000: 22nd International Conference on Software Engineering*, 2000, 5-19.

[12] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice", Keynote Paper*, Proc. RE'04, 12th IEEE Joint Intl. Requirements Engineering Conf.*, Kyoto, Sept. 2004, 4-8.

[13] 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", In *Grammatical Inference*, Lecture Notes in Artificial Intelligence Nr. 1433, Springer-Verlag, 1998, 1-12.

[14] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *Proc. ICSE'02: 24th Intl. Conf. on Soft. Engineering*, Orlando, May 2002.

[15] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Monitoring and Control in Scenario-Based Requirements Analysis", *Proc. ICSE 2005 - 27th Intl. Conf. Software Engineering*, St. Louis, May 2005.

[16] E. Mäkinen and T. Systä, "MAS – An Interactive Synthesizer to Support Behavioral Modelling in UML", *Proc. ICSE'01 – Intl. Conf. Soft. Engineering*,, Toronto, Canada, May 2001.

[17] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. Second Edition, Wiley, 2006.

[18] Z. Manna and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems", Springer-Verlag, 1992.

[19] J. Niu, J.M. Atlee, and N. Day, "Understanding and Comparing Model-Based Specifications Notations", *Proc. RE'03, 11th IEEE Joint Intl. Requirements Engineering Conf.*, Monterey, Sept. 2003, 188-199.

[20] J. Oncina and P. García, "Inferring Regular Languages in Polynomial Update Time", *In* N. Perez de la Blanca et al (Ed.), *Pattern Recognition and Image Analysis*, Vol. 1 Series in Machine Perception & Artificial Intelligence, World Scientific, 1992, 49–61.

[21] S. Uchitel, J. Kramer, and J. Magee, "Detecting Implied Scenarios in Message Sequence Chart Specifications", *Proc. ESEC/FSE'01*, Sept. 2001.

[22] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of Behavioral Models from Scenarios", *IEEE Trans. Softw. Engineering*, 29(2), 2003, 99-115.

[23] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios", *Proc. ICSE'2000: 22nd Intl. Conference on Software Engineering*, Limerick, 2000, 314-323.