Synthesizing Multi-View Models of Software Systems

Bernard Lambeau

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Engineering Sciences

November 2011

Institute of Information & Communication Technologies,
Electronics and Applied Mathematics
(ICTEAM institute)
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgique

Examining board:
Prof. Axel VAN LAMSWEERDE (UCL), Supervisor
Prof. Pierre DUPONT (UCL), Supervisor
Prof. Olivier BONAVENTURE (UCL), Chairperson
Prof. Charles PECHEUR (UCL), Secretary
Prof. Emmanuel LETIER (Univ. College of London)
Prof. Jeff KRAMER (Imperial College London, UK)

Abstract

Models play a significant role for analyzing requirements and exploring designs of software systems. Good models should focus on key aspects of the target system while abstracting from numerous details. The joint use of multiple modeling languages allows analysts to capture complementary system views in a complete, precise, and consistent way. Such models prove difficult to build in practice. Techniques and tools are therefore required for supporting analysts in this task.

Model synthesis techniques may help in building models systematically from various sources of knowledge about the target system. Such techniques may integrate dedicated checks on the models being built. Early errors can thereby be detected and fixed. Model synthesis can also produce missing model fragments, support documentation, or generate code fragments for specific software components.

The thesis proposes tool-supported synthesis techniques for multi-view models involving scenarios, state machines, processes, and goals.

A formal trace semantics is provided for process models together with algorithms for deriving state machines from them. This enables a variety of checks on processes, including the verification that they satisfy desired properties.

An interactive technique is then presented for synthesizing behavior models from scenarios. A grammar induction framework is extended to generalize examples of desired behavior under the control of counterexamples and goals to be met by the system. The induction process interacts with the end-user who needs to classify additional scenarios generated by the synthesizer as positive or negative examples of desired system behavior.

The proposed techniques are implemented by a toolset, whose architecture is briefly described. Our inductive synthesizer has been evaluated in depth on various case studies and on synthetic datasets. These evaluations resulted in an online protocol and platform for evaluating inductive synthesis techniques.

Acknowledgements

Je ne pourrais commencer autrement qu'en remerciant Axel van Lamsweerde, Pierre Dupont et Christophe Damas.

Axel van Lamsweerde a suivi et soutenu mon travail durant pas moins de trois projets de recherche, multipliant ses commentaires, conseils et critiques tant sur le fond que sur la forme, partageant ses propres intuitions, ses idées, son expérience. Sa vision et son enseignement guident aujourd'hui une grande partie de mes connaissances.

Pierre Dupont m'a offert le premier de transformer une bonne intuition de recherche en théorie solide et d'en entrevoir les nombreux avantages. Sa précision, ses encouragements parfois soutenus, son travail et sa disponibilité m'auront accompagné tout au long de notre travail commun en inférence grammaticale, l'un des piliers sur lesquels repose cette thèse.

Christophe Damas est mon collègue depuis 8 ans maintenant, tant dans la recherche que dans l'encadrement des étudiants, lors de déplacements en conférence, face aux arcanes administratives, lors de la rédaction de nos thèses respectives, de leurs présentations. Je lui dois nombre d'enseignements tant sur le plan technique qu'humain.

My examination board deserves special thanks too. Thanks to Olivier Bonaventure, Jeff Kramer, Emmanuel Letier and Charles Pecheur both for their high-level and technical comments. Let me also thank Jeff Magee, Sebastian Uchitel, Dalal Alrajeh and Alessandra Russo from Imperial College London for the fruitful meetings and discussions organized there. On that side of the Channel, special thanks also go to Neil Walkinshaw and Kirill Bogdanov from the University of Sheffield for co-organizing the Stamina competition.

De nombreux collègues ont partagé avec moi leurs avis lors de multiple discussions, dépassant largement le cadre de ce travail voire de l'informatique. Je pense en particulier à José Vander Meulen, François Roucoux, Alain Pirotte, Antoine Cailliau, Renaud De Landtsheer, Jean-Noël Monette, Pierre Schaus, Grégoire Dooms, Stéphane Zampelli, Sébastien Combéfis et Damien Leroy.

Avant tout, je remercie mes parents, mes frères et soeurs, ma compagne Elodie, et sa famille à elle. Pour leur patience, leur intérêt, leur soutien et leurs nombreux mots d'encouragements bien sûr, mais en fait et surtout pour tout le reste.

Ce travail a été soutenu financièrement par la Région Wallone lors des projets ReQuest (Programme WIST 1.0, RW Conv. 315592), Gisele (Programme WIST 2.0, RW Conv. 616425) et PIPAS (Programme WIST 3.0, RW Conv. 1017087) ainsi que par le programme PAI du gouvernement fédéral belge (Pôles d'attraction Interuniversitaire, projet MoVES).

Contents

List of Figures

Chapter 1

Introduction

This chapter introduces the context, motivations, objectives and contributions of the thesis.

Section 1.1 discusses the role of models in software system development. Section 1.2 discusses specific requirements on modeling languages for automated support. Section 1.3 discusses the role of model synthesis in such support. Section 1.4 introduces the contributions of the thesis. Section 1.5 presents the structure of the dissertation.

1.1 Models and their use in software system development

Models are increasingly recognized as an effective means for elaborating requirements and exploring designs of software systems. A *model* is an abstract representation of the target system, where key features are highlighted, specified and inter-related to each other [van09]. Models are widely used at different steps of the software development process and for different software-related purposes.

Requirements engineering, in particular, aims at precisely determining what the system should do and why it should do so. This has been shown to be the hardest part of software development [Bro87]. Using models for exploring, analyzing and specifying requirements has multiple benefits:

- Models force stakeholders and analysts to be more precise in their formulations.
- They allow them to abstract from multiple details in order to focus on key system aspects.
- They provide a basis for early detection and fixing of errors.

Model-oriented approaches to requirements elaboration include KAOS [van09] and NFR/i* [Myl92, Yu,93]. The models there capture how system goals may be refined in lower-level goals and how the latter are operationalized in operations and constraints. *Goals* in such frameworks capture prescriptive statements of intent to be satisfied by the agents forming the system. KAOS puts emphasis more on semi-formal and formal reasoning about behavioral goals. These are goals prescribing system behaviors declaratively. In NFR/i*, the emphasis is more on qualitative reasoning on soft goals; these are goals that cannot be established in a clear-cut sense.

Software design may be driven by Model Driven Engineering approaches (MDE), often based on UML models [Obj04] or Domain Specific Languages [van00a, Mar10]. Such approaches model the application domain rather than the computing elements. The use of models for software design has multiple benefits too:

- Models enforce the separation between abstract domain concepts and their concrete implementation. They allow engineers to focus on each aspect at the adequate level of abstraction.
- Code fragments can often be generated from models; this appears less error-prone and time consuming than manually translating models to code. In case of domain-specific languages, models may even be already executable.
- Design models also yield a natural documentation for the software and the underlying design decisions.

On another hand, *work processes* may be captured by graphical models. Such models take various forms, e.g., flowchart-style models such as Activity Diagrams [Obj04], YAWL [Van05] or diagrams in the Business Process Modeling Notation (BPMN) [Obj08] or task trees, e.g. Little-Jil [Cla08]. Modeling here proves useful too.

- Process models may be analyzed and verified before translating them to work organization on the field. Early errors and bottlenecks may thereby be detected.
- Model analysis may also result in restructuring work organization for better performance.
- Process models may be partly or fully enacted to trigger the execution of necessary tasks, issue reminders to process actors, control the automated scheduling and processing of tasks.

1.2 Requirements on modeling languages

Complex systems are better described through multiple views. Different models focus on different facets of the system along its intentional, structural, operational and behavioral dimensions [Rum91, Fin92, van09].

Scenarios, for example, may illustrate typical interactions among agent instances whereas *state machines* capture a complete specification of agent behaviors. A *process model* may describe the operational decomposition of a complex work into smaller tasks. *Goals* may make the system objectives precise and articulate them from high-level concerns to fine-grained details.

Building high-quality models in such multi-view approaches is far from easy. Suitable modeling languages, tools and techniques are therefore required to help analysts in this task. To play a significant role, a good multi-view modeling language should meet the following requirements [van09]:

- *Multi-level*: the system should be captured at different levels of abstraction and precision to enable stepwise elaboration and validation;
- *Analyzable*: the modeling abstractions should be accurate enough to support useful forms of analysis.

In addition, a modeling language should support stakeholders and analysts in building high-quality models. A "good" multi-view model should meet the following requirements:

- *Adequate*: the models should adequately represent the essence of the target system while abstracting from unnecessary details;
- *Complete*: the models should capture all pertinent facets of the system along the WHY-, WHAT- and HOW- dimensions;
- *Precise*: the models should be accurate enough to capture system descriptions with as little ambiguity as possible;
- *Consistent*: the models should agree on their overlapping descriptions of the system;
- *Comprehensible*: the models should be easy enough to understand by the people who need to use them.

Formal modeling approaches help meeting the above requirements. The semantics of a formal modeling language provides precise rules of interpretation that allow many of the problems with natural language to be overcome.

Formal specifications may also be manipulated by automated tools for a wide variety of purposes.

In a disciplined approach, model *analysis* and model *synthesis* are intertwined activities aimed at reaching high-quality models, especially towards increased completeness, consistency and precision.

This thesis investigates model synthesis. A companion thesis by Christophe Damas investigates model analysis more specifically [Dam11].

1.3 Synthesis as a promising approach for multi-view model building

Model synthesis can be roughly defined as the systematic construction of models from various sources of knowledge about the target system.

Common sources of knowledge for model building include early system descriptions, interviews with stakeholders, software prototypes, etc. For an in-depth description of requirements elicitation and system modeling from such sources see, e.g., [van09].

Multi-view modeling frameworks offer new opportunities for effectively supporting the incremental building of models. In particular, they often come with precise rules of consistency between the models involved. These rules are used for automating consistency checks. The same rules can also be used the other way round, to semi-automatically synthesize model fragments using the information already available in other views.

The use of synthesis techniques may be motivated by various needs. As a consequence, the term "model synthesis" may cover a broad spectrum of techniques and practices.

In this thesis, we will focus on two particular and complementary forms of synthesis:

Horizontal synthesis is aimed at completing a multi-view model of the target system. A synthesis technique is used here to build model fragments missing from a multi-view framework or to complete existing ones.

Vertical synthesis is aimed at deriving lower-level models from higher-level ones. It thereby provides an operational semantics for high-level models and makes model-checking tools available to them.

These two forms of model synthesis will be seen to be fairly different in the thesis.

- In vertical synthesis, the synthesized models are defined at a lower level of abstraction than the source models they come from; this is not necessarily the case in horizontal synthesis.
- The analyses enabled by vertical synthesis are performed on models having a shorter lifetime – generally limited to the analysis itself. In horizontal synthesis, the resulting models are often kept as software documentation or for requirements traceability.
- Vertical synthesis is derivational by nature whereas horizontal synthesis may be inductive.
- In vertical synthesis, models are generally kept hidden from the user; the latter has a passive role in the synthesis process. In horizontal synthesis, the produced models must generally be validated by the end-user. The latter may even play an active role in the synthesis process itself.

The core chapters of this thesis will cover those two forms of model synthesis.

- A technique will be presented for synthesizing agent behavior models from scenarios. The language of Message Sequence Charts (MSCs) will be chosen for describing scenarios of agent interactions. Agent behaviors will be modeled with LTS state machines. The semantic links between MSC scenarios and LTS state machines is borrowed from [Uch03].

  The proposed synthesis will rely on grammar induction [Onc92, Lan98]. Our inductive synthesis techniques will take scenarios as examples and counterexamples of desired system behavior; user interactions will be added to accept or reject scenarios generated by the technique. Moreover, domain knowledge such as goals or models of legacy components will be taken into account to ensure the consistency of the synthesized models while speeding-up the induction process and reducing the number of user interactions.

- Guarded high-level Message Sequence Charts (g-hMSC) are introduced in [Dam10, Dam11] to model critical processes such as the ones involved in medical workflows. Such models take the form of flowcharts of tasks and decision nodes with outgoing guarded transitions labeled with Boolean expressions on process variables.

  In the thesis, a precise trace semantics for g-hMSC will be defined in terms of labeled transition systems (LTS) [Kel76, MK99]. To achieve this, a guarded flavor of LTS is introduced as an intermediate model. Synthesis algorithms from g-hMSC to guarded LTS (g-LTS) to LTS

will be described. These algorithms make LTS-based model checking available to g-hMSC process models though slight adaptations of a technique borrowed from [Gia03].

1.4 Overview of contributions

This section summarizes the main contributions of the thesis. Section 1.4.1 lists contributions related to grammar induction. They provide a basis for the inductive synthesis of state machines from scenarios. The contributions along this horizontal axis of model synthesis are summarized in Section 1.4.2. Section 1.4.3 lists those related to the vertical synthesis of LTS behavior models from process models.

1.4.1 Extended techniques for grammar induction

- The thesis presents a new induction algorithm called Query-Driven State Merging (QSM). This algorithm is an interactive extension of state-of-the-art algorithms in grammar induction, namely RPNI and Blue-fringe [Onc92, Lan98]. QSM extends those algorithms with an oracle – typically, an end-user. The oracle interacts with the induction process by answering so-called "membership queries", that is, by classifying generated strings as positive or negative elements of the induced regular language.

- The thesis presents the Automaton State Merging (ASM) algorithm as an additional induction technique. This algorithm allows generalizing a positive *language* as opposed to a positive *sample* – that is, a finite set of induction strings. The generalization process is made under the control of a negative sample.

- The thesis describes a new protocol for evaluating grammar induction algorithms, called Stamina. This protocol is an alternative to the existing Abbadingo procedure [Lan98]; it has been designed for samples defined on larger alphabets and obtained from the target automaton itself. Samples and automata considered in Stamina are more representative of the scenarios and state machines encountered in software engineering models.

1.4.2 Horizontal synthesis of behavior models

- The thesis provides algorithms for inductively synthesizing state machines models from MSC scenarios and high-level MSCs. These algorithms rely on QSM and ASM to generalize the system behaviors

illustrated in scenarios. QSM is used to synthesize state machines from collections of MSCs; the end-user interacts with the synthesis process by classifying generated scenario questions as examples or counterexamples of desired system behavior. ASM is used to synthesize state machines from high-level MSCs.

- The thesis shows how various sources of knowledge about the target system can be incorporated to constrain the synthesis process so as to ensure the consistency of the synthesized state machines with that knowledge. A general constraint mechanism is instantiated to inject knowledge about state variables, goals, and domain properties, and behavior models of legacy components. The injection of such knowledge prunes the induction search space, thereby speeding-up the synthesis process and reducing the number of user interactions.

- The thesis provides in-depth evaluations of QSM, ASM, and our resulting inductive synthesis techniques. The evaluations were conducted on common case studies from the literature and complemented with experiments on synthetic datasets. The benefits of introducing user interactions through scenario questions and of injecting system knowledge in the induction process are illustrated and quantified in both cases.

- A tool supporting our inductive synthesis approach is also described. This tool was successfully used for the evaluation of our approach on case studies.

1.4.3 Vertical synthesis of formal process models

- The thesis provides a formal trace semantics for the g-hMSC process language described in [Dam11]. This trace semantics is defined in terms of guarded LTS; it supports the model-checking of process models thanks to a trace-based model checker adapted from LTSA [MK99].

- The thesis extends LTS state machines with guarded transitions defined on fluents. It provides a declarative trace semantics for such guarded LTS. It also proposes a composition operator for them together with an algorithm for capturing their semantics as pure LTS.

- The thesis describes the implementation of a compositional model checker for g-hMSC and g-LTS models. It also describes key aspects in the implementation of a toolset aimed at supporting the modeling of guarded processes and the analysis of such models.

1.5 Organization of the thesis

The thesis is structured as follows.

Chapter 2 sets the background picture for the thesis as a guided tour of the multi-view modeling framework considered. It describes the formalisms used for capturing scenarios, state machines, goals and process models. The chapter also states important consistency rules between these specific views.

Chapter 3 describes our technique for vertical synthesis of labeled transition system from process models. A trace semantics for process models is defined together with algorithms for deriving lower-level models. The first algorithm maps process models into guarded LTS; the second algorithm maps the latter into pure event-based LTS.

Chapter 4 describes our inductive techniques for horizontal synthesis of state machines from scenarios, based on interactive grammar induction. From a minimal problem statement, constraints on the synthesis process are gradually added and the algorithms are incrementally enhanced to meet these constraints.

Chapter 5 discusses how our inductive synthesis technique has been evaluated. It covers both evaluations on case studies and evaluations on synthetic datasets.

Chapter 6 discusses Stamina, our online platform for evaluating inductive synthesis techniques. Stamina first ran as a formal competition whose results are also briefly summarized.

Chapter 7 discusses the toolset developed to support the proposed techniques, namely, a model checker for g-hMSC process models, an interactive synthesizer of state machines from scenarios, and a tool for modeling and analyzing process models for critical applications such as complex medical workflows.

Chapter 8 reviews related work, discussing how other synthesis techniques complement and/or differ from those proposed in the thesis.

Chapter 9 concludes the thesis and discusses open issues, perspectives, and future work.

Chapter 2

A Multi-view Framework for System Modeling

This chapter introduces the necessary background on modeling formalisms and techniques used in the thesis. It is organized as follows. The running examples used throughout the thesis are briefly presented in the next section. Section 2.2 then provides a general overview of the modeling framework, its main hypotheses, and how multiple models fit together. The models and their semantics are then detailed in subsequent sections. Section 2.3 introduces the class of state machines considered. Scenarios and related constructs are defined in Section 2.4. To integrate event-based and state-based specifications, fluents are introduced in Section 2.5. The fluent-based specification of goals and domain properties is discussed in Section 2.6. Section 2.7 introduces guarded process models.

2.1 Running examples

This section introduces the two running examples that will be used throughout the thesis. Section 2.1.1 introduces a simple train control system. Section 2.1.2 introduces a meeting scheduler system [Fea97].

2.1.1 A simple train control system

A simplified train control system will be used as a running example for illustrating behavioral model concepts and techniques throughout the thesis.

The system is composed of a software train controller, actuators for doors and train acceleration, sensors and passengers. Through the actuators, the software controller controls operations like starting or stopping the train, opening or closing doors, and so on. A safety goal requires train doors to

remain closed while the train is moving. If the train is not moving and the passenger presses the alarm button, the controller must open the doors immediately. If the train is moving and the passenger presses the alarm button, the controller must stop the train first and then open the doors.

The latter situation is illustrated with a scenario in Fig. 2.1, where horizontal arrows denote interaction events among agent instances. The precise semantics of such scenario will be made clear in the following sections.

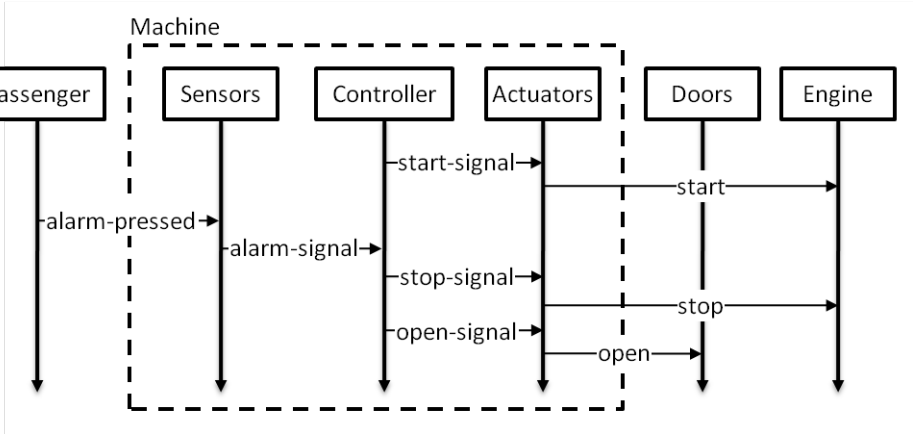


Figure 2.1: A scenario illustrating train stopping in emergency when an alarm is pressed.

2.1.2 The meeting scheduler

For illustrating process models and associated techniques, we will focus on the following episode of a meeting scheduling process [Fea97].

A meeting initiator issues a meeting request, specifying the expected participants and the date range within which the meeting should take place. The scheduler then sends an electronic invitation to each participant, requesting them to provide their date constraints.

A date conflict occurs when no date can be found that fits all participant constraints. In such case, the initiator may extend the date range or request some participants to weaken their constraints; a new scheduling cycle is then required. Otherwise, the meeting is planned at a date meeting all constraints.

A soft goal requires meetings to be scheduled as quickly as possible once initiated; another one requires interactions with participants to be minimized. In the simplified version considered here, only two scheduling cycles are allowed; the meeting is automatically planned after that. In such case, we will assume that the best date is chosen so as to maximize

the number of participants attending. We also ignore features like meeting cancellations, meeting locations, and so on.

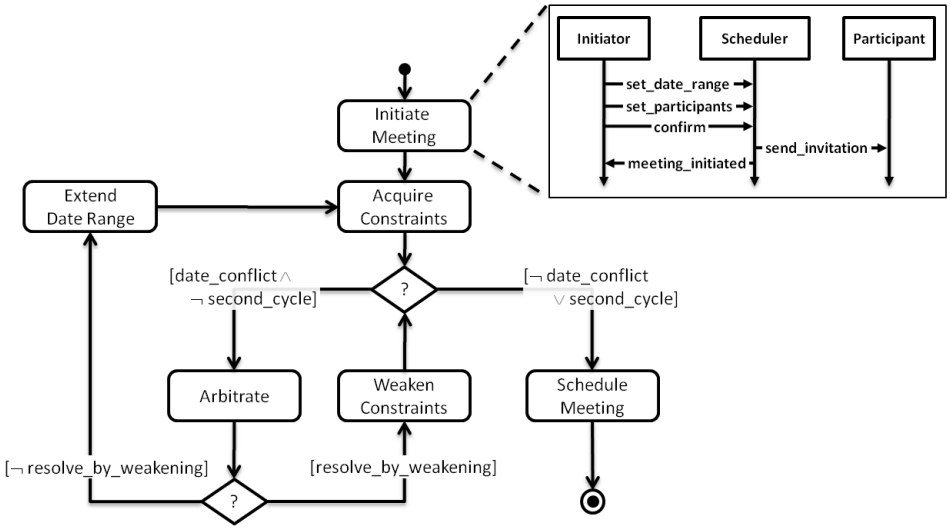


Figure 2.2: A process model for a meeting scheduling process.

This meeting scheduling process is illustrated in Fig. 2.2, where boxes denote tasks. The precise semantics of such model will be detailed in Chapter 3.

2.2 Framework overview

Modeling software systems calls for rich models that cover the structural, intentional and behavioral dimensions of the system [van00b]. Our framework integrates these co-related dimensions with an emphasis on the behavioral one. We start with a guided tour along those dimensions before discussing how multiple models actually fit together in a coherent way.

2.2.1 Capturing multiple views of the system

The structural view – A system is commonly seen as being made of active components, called *agents*, that behave and interact so as to fulfill system goals; they need to restrict their behavior so as to ensure the goals they are responsible for [Fea87]. Some of them are human agents (e.g. passenger in Fig.2.1); others are physical or electronic devices (e.g. train doors and actuators); others are software components (the software controller).

In addition to the notion of *system*, that encompasses all agents, the literature makes use of specific terms to distinguish between certain agents and/or agent aggregations. In [van09] for example, the *software-to-be* denotes software agent(s) that need to be developed (the automated controller, for example) while the other agents compose its *environment*. Another boundary consists in distinguishing the software together with its input and output devices from the other agents. This boundary, depicted with a dashed line in Fig. 2.1, corresponds to the distinction between the *world* and the *machine* [Jac95].

The thesis sticks to very basic notions of structural modeling. The interfaces among the agents composing the system consist of messages, called *events*, that the agents can send or receive (see the behavioral dimension below). For sake of simplicity we assume that event labels uniquely determine agent interactions. We do not consider specific abstractions for modeling agent interfaces and boundaries, like context diagrams [Jac95] and events structured in terms of attributes.

As illustrated by the *Machine* boundary in Fig. 2.1, a set of agents can be aggregated into a new one of coarser granularity. Events can then be partitioned among internal, boundary and external events. The boundary events form the new agent's interface. The surrounding box can be seen as a white or a black-box dependent on whether internal events are shown or hidden, respectively. The composition and hiding operators on state machines support such structural mechanisms on the behavioral side (see Section 2.3). For a more precise description of structural models that nicely fit our framework the reader can refer to [Mag95].

The behavioral view – Behaviors capture the dynamic interactions among the agents forming the system; they will be modeled as sequences of events. In an interaction, an event is *synchronously* sent by a source agent and received by a target agent. The same event can in fact be received by several agents at once; a form of *broadcasting* is thus supported.

Typical examples and counterexamples of system behavior are specified through positive and negative scenarios involving agent instances, like the one in Fig. 2.1. The Message Sequence Charts (MSC) notation will be used to capture scenarios [ITU96]. Higher-level scenarios will be supported by introducing sequences and loops in such descriptions.

In addition to the partial behavior description linking agent instances, the complete behavior of each agent will be modeled at class level through a form of state machine known as labeled transition systems (LTS) [Kel76, Mil89]. The behavior of the entire system is obtained by parallel composition [Hoa85] of the agent LTSs. Behavior projection on specific agents is also supported, in order to get the zoom-in/zoom-out facilities as suggested before.

This thesis will focus on *determinate* agents [Eng85]; these are agents whose observable behavior can be captured with the sole use of *deterministic* transition systems (see Section 2.3).

- This restriction results in a simple and intuitive framework, for easier accessibility to stakeholders involved in the early phases of system design.
- It also allows us to formalize behaviors with standard *trace theory* [Hoa85] and stick ourselves to the simplest notion of behavior equivalence, namely *trace equivalence* [Eng85].
- Agent and system behaviors can then be captured by the class of *prefix-closed* regular languages, a subclass of the well-studied *regular* languages [Hop79, Aho86]. Further to enabling the reuse of standard results from automaton theory, this paves the way to the use of grammar inference [Gol78] for behavior model synthesis (see Chapter 4).

The intentional view – This view is aimed at capturing *why* the system is needed. A *goal* is a prescriptive statement of intent whose satisfaction requires the collaboration of system agents. Unlike goals, *domain properties* are descriptive statement about the environment – such as physical laws, organizational rules, etc. Goal models are AND/OR graphs that capture how functional and non-functional goals contribute positively or negatively to each other [van00b, van04].

The thesis will focus on behavioral goals. A *behavioral* goal implicitly defines a maximal set of admissible system behaviors. Unlike *soft* goals, behavioral goals can be established in a clear-sense (see [van09] for a taxonomy of goals). For model synthesis the thesis will only consider goals and domain properties that can be formalized as *safety* properties in linear temporal logic (LTL) [Man92]. A safety property stipulates that some "bad thing" may never happen. If such a "bad thing" happens in an infinite sequence, then it must also do so after some finite prefix and must be irremediable [Alp86, Gia99]. Interestingly, the class of system behaviors satisfying safety properties can be expressed through labeled transition systems [Gia03]; behavioral goals and domain properties can therefore be integrated in our framework without much additional machinery.

For the modeler, goals are best captured through state-based abstractions (e.g. "the train may be *moving* only if the doors are *closed*"). In contrast, the formal behavioral view is event-based (e.g. "the controller issues a *start* command after having issued a *close-doors* command").

Fluents will be used as an effective means for reconciling these two para-digms [Mil02]; they capture state-based propositions in terms of the occurrence of events (see Section 2.5). Behavioral goals will therefore be

formalized in Fluent Linear Temporal Logic (FLTL) [Gia03], a flavor of linear temporal logic where atomic propositions are fluents. The structuring of goals in goal graphs and their assignment to agents will not be considered in the thesis.

The operational view – This view aims at capturing processes which agents are involved in. Processes will be made of tasks and decisions. A *task* is a unit of work to be performed by collaboration of the agents. A *decision* is a condition on the process state which drives which tasks are to be performed next.

The guarded high-level Message Sequence Charts (g-hMSC) notation will be used to capture processes [Dam09, Dam11]. Tasks in a g-hMSC are either MSC scenarios or finer-grained g-hMSCs. Decision nodes capture conditions on the process state in terms of fluents. The semantics of g-hMSC will be defined in Chapter 3.

2.2.2 On the double use of labeled transition systems

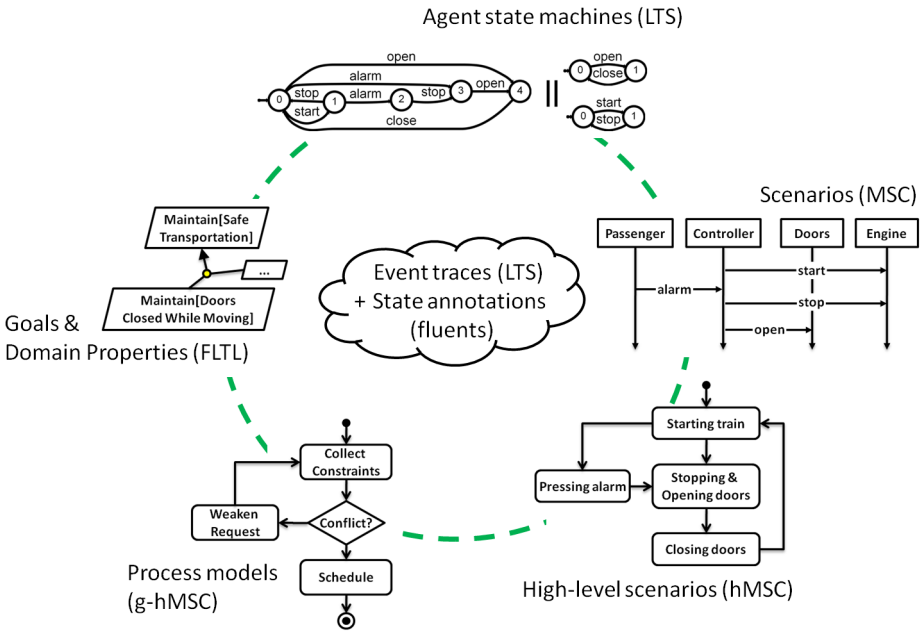


Figure 2.3: Formal framework outline.

The multi-view framework is depicted in Fig. 2.3. This figure provides a key for reading the rest of this chapter. As we put more emphasis on behavior modeling, all models will be grounded on standard trace theory [Hoa85]. *Traces* are finite sequences of event labels. They will be used to capture

the system behaviors described in a MSC scenario, those admitted by a behavioral goal, and so on. Labeled transition systems (LTS) will allow us to capture and manipulate *sets* of traces. A trace-based semantics will provide precise answers to questions such as:

- What agent and system traces does this scenario cover?
- Is this trace accepted by this state machine?
- Does this sequence of events violate this behavioral goal?

Note that labeled transition systems appear twice in the figure as we will make two different uses of them:

- They provide a representation for the sets of traces that capture the model semantics.
- They are also chosen as a particular representation for agent state machines[1].

This double use of LTS illustrates the two different perspectives on behavior model synthesis considered in the thesis:

- In Chapter 3, LTS synthesis is used to capture the semantics of process models and make the latter analyzable;
- In Chapter 4, LTS synthesis is used to infer the state machines of system agents from scenarios illustrating interactions among them.

2.3 State machines as labeled transition systems

In our framework, the behavior of an agent will be modeled by a specific kind of finite state machine called *labeled transition system* (LTS). This formalism, initially introduced by Keller for reasoning about parallel programs [Kel76], has been intensively used for specifying and analyzing concurrent systems, e.g. in [Mil89, Cla89, Mag97].

A LTS is made of a set of states and a set of transitions between them (see Fig. 2.4). Each transition has an *event* label – sometimes called *symbol* or *action* label. A state is labeled with a unique number to distinguish it from other states. An *initial state* is denoted graphically by an incoming arrow with no source state (e.g. state 0 in Fig. 2.4).

[1] Even though one could argue that such representation is too low-level to be accessible to stakeholders.

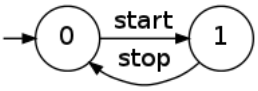


Figure 2.4: A Labeled Transition System for an `Engine` agent.

Definition 2.1 (Labeled Transition System)**.** *A LTS is defined as a 4-tuple* $(Q, \Sigma, \delta, q_{init})$ *where*

- Q *is a finite set of states,*
- Σ *is a set of labels called its* alphabet*,*
- δ *is a transition relation* $Q \times \Sigma \cup \{\tau\} \times Q$*,*
- $q_{init} \in Q$ *is the initial state.*

The *alphabet* Σ captures the notion of *agent interface* as a set of event labels that an agent recognizes. These are the events in which the agent *engages* in synchronous communications with its environment. For example, the LTS in Fig. 2.4 has an alphabet $\Sigma = \{start, stop\}$.

The set of finite sequences over an alphabet Σ will be denoted by Σ^*.

The label τ in the above definition of a LTS is used to denote so-called *non-observable* transitions. Such transitions allow us to model agent state changes that cannot be monitored by its environment. In the sequel, we will denote $\Sigma \cup \{\tau\}$ by Σ_τ, referring then to an alphabet augmented with the τ label.

Note that LTS do not distinguish between *sent* and *received* events. Such distinction may be required when connecting state machines with scenarios. We will assume this structural information to be available elsewhere, typically from a context or architecture diagram [War85, Mag95]. We will also assume that an event label uniquely determines the interacting agents; an event may however be received by more than one of them.

Definition 2.2 (Trace)**.** *Given an alphabet* Σ*, a* trace *is an element of* Σ^**, that is, a finite sequence of event labels* $w = <l_0, \ldots, l_n>$ *with* $l_i \in \Sigma$*. We will sometimes use the notation* vw *to denote the concatenation of a trace* v *with another trace* w*.*

Definition 2.3 (Deterministic LTS)**.** *A LTS is* deterministic *if a trace always uniquely determines the reached state; otherwise it is* non-deterministic*. A deterministic LTS may therefore not have* τ *transitions; neither does it have a state with two outgoing transitions sharing the same label, that is,*

$$(q, l, q_1) \in \delta \wedge (q, l, q_2) \in \delta \implies q_1 = q_2$$

Definition 2.4 (Terminating LTS)**.** *A state is* terminating *if it has no outgoing transition; otherwise it is* non-terminating*. A* terminating *LTS has at least one terminating state; otherwise it is* non-terminating*.*

Note that the above definition does not allow distinguishing between terminating states that capture successful termination (where an agent stops running intentionally) and non-successful termination (e.g., an agent composed from finer-grained agents *deadlocks* unintentionally). Deadlock analysis will be left outside the scope of the thesis.

A *trace semantics* is used in this thesis for capturing agent behaviors, in terms of the set of traces that they accept [Hoa85]. Additional definitions are needed here.

Definition 2.5 (LTS execution)**.** *A* LTS execution *is a finite sequence of states separated by labels:*

$$w = <q_0, l_0, \ldots, q_{n-1}, l_{n-1}, q_n>$$

with $q_i \in Q$ *and* $l_i \in \Sigma_\tau$*.*

The *projection* of an execution w over an alphabet Σ, denoted by $w|_\Sigma$, is the result of keeping from w only those event labels that belong to Σ – in other words, q_i states and τ labels have been eliminated. The projection of an execution yields a trace.

Definition 2.6 (Valid LTS execution)**.** *An execution is* valid *for a LTS if it denotes a path from the initial state in the corresponding graph:*

$$q_0 = q_{init} \text{ and } (q_i, l_i, q_{i+1}) \in \delta \text{ for } 0 \leq i < n.$$

Definition 2.7 (Accepted trace)**.** *A trace* t *is* accepted *by a LTS if there exists a valid execution* w *such that* $w|_\Sigma = t$*.*

In other words, a trace is accepted by a LTS if it denotes an existing path in the corresponding graph from the initial state, possibly with silent moves offered by τ transitions in the non-deterministic case. In the latter case, observe that a trace may actually denote more that one existing path.

As a consequence of this definition, a prefix of an accepted trace is an accepted trace as well; the empty trace λ is always accepted.

For example, the LTS in Fig. 2.4 accepts the trace `<start stop start>`, the trace `<start stop>`, but not the trace `<start start>`.

Definition 2.8 (Language of a LTS)**.** *The set of traces accepted by a LTS* P *is called its* language*; it is denoted by* $\mathcal{L}(P)$*.*

For example, the language of the LTS shown in Fig. 2.4 is

$$\mathcal{L}(\texttt{Engine}) = \{\lambda, \texttt{<start>}, \texttt{<start stop>}, \texttt{<start stop start>}, \dots\}$$

Behavioral equivalence is an important notion when designing and analyzing concurrent systems. This notion actually drives the behavioral semantics by answering questions such as "*are agents Ag_1 and Ag_2 the same in terms of their observable behavior?*". Many different notions of behavioral equivalence exist in the literature, like *strong* and *observational* equivalences [Mil89], bisimilarity [Par81], or failure equivalence [Hoa85].

In this thesis, we assume *determinate* agents, that is, agents whose observable behavior can be captured with the sole use of a deterministic transition system [Eng85]. This allows us to stick to the weakest and simplest notion of LTS equivalence, namely trace equivalence [Hoa85, Eng85].

Definition 2.9 (Trace equivalence)**.** *Two LTS P and Q are* trace-equivalent *if they accept the same set of traces:*

$$P \equiv_{tr} Q \text{ if and only if } \mathcal{L}(P) = \mathcal{L}(Q)$$

Under trace equivalence, LTS inherit operators and properties from their connection with regular languages. Some important results are summarized in the next section.

2.3.1 Labeled transition systems and regular languages

Labeled transition systems are actually a subclass of finite automata [Hop79].

Definition 2.10 (Finite automaton)**.** *A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_{init}, F)$ where*

- *Q is a finite set of states,*
- *Σ is an alphabet,*
- *δ is a transition relation $Q \times \Sigma \cup \{\tau\} \times Q$,*
- *q_{init} is the initial state, and*
- *F is a subset of Q identifying the accepting states.*

The only difference between LTS and finite automata is that the latter distinguish between *accepting* states ($F \subseteq Q$) and *non-accepting* ones ($Q \setminus$

F). Accepting states will be depicted with double circles. Unlike LTS, only traces that end in an accepting state are accepted by a finite automaton.

A finite automaton is shown in Fig. 2.5. State 0 is accepting whereas state 1 is non-accepting. As a consequence, this automaton accepts the trace `<a b b>` but not the trace `<a b>`.

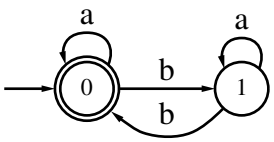


Figure 2.5: A finite automaton.

A LTS is a standard automaton in which all states are accepting, that is $F = Q$. Our choice of trace equivalence allows us to make use of many results from standard automata theory. In particular,

- Standard automata, both deterministic and non-deterministic, capture the well-studied class of *regular* languages [Hop79]. As they only have accepting states, LTS capture the subclass of *prefix-closed* regular languages. The term "prefix-closed" means that all prefixes of accepted traces are also accepted traces:

$$prefixes(\mathcal{L}(P)) = \mathcal{L}(\mathcal{P})$$

- For any regular language $\mathcal{L}$, there exists a canonical automaton $A(\mathcal{L})$. This automaton is the minimal deterministic automaton accepting $\mathcal{L}$; it is known to be unique up to state renumbering [Hop79].

Without loss of generality, we may therefore assume that the behavior of any agent is modeled by a canonical LTS. Moreover, we may use non-deterministic constructions, τ transitions in particular, without contradicting the assumption of *determinate* agents.

Given a LTS P, deterministic or not, we will denote by P^{Δ} its canonical and deterministic equivalent. In the latter notation, P can actually be a LTS *expression*, that is, a LTS obtained by the application of the LTS operators introduced in the next sections. Standard automaton algorithms from [Hop79] can be used to compute P^{Δ}. This typically involves removing τ transitions, determinizing and minimizing the LTS under trace equivalence.

As languages capture sets of traces, it is sometimes convenient to reason in terms of standard operators on sets. In the following sections, we will

often make use of notations for the union of two languages ($\cup$), their intersection ($\cap$), subset ($\subseteq$), proper subset ($\subset$) and equality ($=$). Techniques and algorithms for implementing these operators for the general case of standard automata can be found in [Hop79, Aho86].

For a given language $\mathcal{L}$, $mt(\mathcal{L})$ will denote the set of *maximal* traces of $\mathcal{L}$, that is, traces that cannot be extended by a suffix within the same language. In the case of a deterministic LTS, they simply correspond to traces reaching a terminating state (see Def. 2.4).

The next two sections define two additional operators on LTS, namely *composition* and *hiding*. They support reasoning about behaviors in presence of multiple agents with different alphabets.

2.3.2 Systems as agent compositions

A system is composed of active agents whose behavior is explicitly modeled by a LTS. The behavior of the system itself is defined through parallel composition [Hoa85]. In this setting, agents execute asynchronously but synchronize on shared events. A system made of n agents is defined as:

$$System = Ag_1 \parallel \ldots \parallel Ag_n \tag{2.1}$$

As we are mostly interested in agent *behaviors*, for parallel composition we will use the binary composition operator $\parallel$ defined on LTS [Gia99, MK99].

The trace semantics of a system composed of n agents whose behavior is modeled with LTSs $Ag_1 \ldots Ag_n$ will then be captured by:

$$\mathcal{L}(System) = \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) \tag{2.2}$$

The LTS composition operator computes the interleaving of all traces accepted by the two LTS under the constraint that they synchronize on shared labels. This operator is both commutative and associative, allowing us to write (2.1) and (2.2) without ambiguity.

Definition 2.11 (LTS composition)**.** *Let* $P = (S_1, \Sigma_1, \delta_1, q_1)$ *and* $Q = (S_2, \Sigma_2, \delta_2, q_2)$ *denote two LTS. Their* composition *define the following LTS:*

$$P \parallel Q = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \delta, (q_1, q_2))$$

where δ *is the smallest relation satisfying the following rules [Gia99]:*

$$\frac{P \xrightarrow{l} P'}{P \parallel Q \xrightarrow{l} P' \parallel Q} \; l \notin \Sigma_2 \qquad \frac{Q \xrightarrow{l} Q'}{P \parallel Q \xrightarrow{l} P \parallel Q'} \; l \notin \Sigma_1$$

$$\frac{P \xrightarrow{l} P',\ Q \xrightarrow{l} Q'}{P \parallel Q \xrightarrow{l} P' \parallel Q'} \; l \neq \tau$$

The notation $X \xrightarrow{l} X'$ *means that the LTS* $X = (S, \Sigma, \delta, q_0)$ *may* transit *into another LTS* $X' = (S, \Sigma, \delta, q_1)$ *through the event label* l*, provided that* $(q_0, l, q_1) \in \delta$.

$P \parallel Q$ is thus defined on the Cartesian product of the sets of states of P and Q; its initial state is the pair (q_1, q_2). The above rules define the possible transitions from such a state.

- The first two rules are symmetric; they encode the fact that, on non-shared labels, one LTS may transit while the other stays in its previous state. Those rules also allow individual LTS to move along their τ transitions.
- The last rule forces the two LTS to transit together on all shared labels but τ.

A composed LTS can be easily computed in a constructive way by exploring the state space from its initial state until no new state pair is discovered.

Note the following relation between the traces accepted by the composition of two LTS $P = (S_1, \Sigma_1, \delta_1, q_1)$ and $Q = (S_2, \Sigma_2, \delta_2, q_2)$ and the traces accepted by each of them individually:

$$\forall s \in \mathcal{L}(P \parallel Q),\ s|_{\Sigma_1} \in \mathcal{L}(P) \text{ and } s|_{\Sigma_2} \in \mathcal{L}(Q)$$

In other words, projecting the traces accepted by a composed LTS on the respective alphabets of its two components yields traces accepted by the latter. It follows that the composition operator computes the intersection of accepted traces when the two LTS operands share the same alphabet:

$$\mathcal{L}(P \parallel Q) = \mathcal{L}(P) \cap \mathcal{L}(Q) \quad \text{if} \quad \Sigma_1 = \Sigma_2$$

2.3.3 Black-box behavior through event *hiding*

It is sometimes useful to consider the composition of a subset of agents that together define an interesting boundary in the system considered.

Consider the agents depicted in the scenario of Fig. 2.1, for example. The machine boundary can simply be modeled as follows:

$$\begin{aligned} Machine &= Controller \parallel Actuators \parallel Sensors \\ World &= Passenger \parallel Doors \parallel Engine \\ System &= Machine \parallel World \end{aligned}$$

Following Definition 2.11 of LTS composition, the alphabet of the *Machine* agent above is $\Sigma_{Controller} \cup \Sigma_{Actuators} \cup \Sigma_{Sensors}$. The alphabet

captures the sets of events in which the agent engages and is therefore too large here. A black-box version of the *Machine* behavior might be more suitable; it is a LTS whose interface is restricted to those events crossing the depicted boundary. The hiding operator will be used to restrict agent interfaces in such cases.

Definition 2.12 (LTS hiding)**.** *The* hiding *of a set of labels I in a LTS $P = (Q, \Sigma, \delta, q_{init})$ defines the LTS:*

$$P \setminus I = (Q, \Sigma \setminus I, \delta_{hidden}, q_{init})$$

where δ_{hidden} is the smallest relation satisfying the following rules [Gia99]:

$$\frac{P \xrightarrow{l} P'}{P \setminus I \xrightarrow{l} P' \setminus I} \; l \notin I \qquad \frac{P \xrightarrow{l} P'}{P \setminus I \xrightarrow{\tau} P' \setminus I} \; l \in I$$

The hiding operator thus makes a set of labels invisible to the environment by replacing them by τ transitions. The resulting LTS is thus non-deterministic. However, a minimal and deterministic LTS that is trace equivalent is guaranteed to exist (see Section 2.3.1).

In our example, a suitable LTS for the black-box machine is given by:

$$Machine' = (Machine \setminus Internals)^{\Delta} \; ,$$

where *Machine* is the composition between the controller, actuators and sensors given previously, and *Internals* is the set of events internal to the machine – here, {`start-signal`, `stop-signal`, `open-signal`, ...} (see Fig. 2.1).

Given a LTS $P = (Q, \Sigma, \delta, q_{init})$ and a set of labels I, the relation between the languages of P and $P \setminus I$ is defined as follows:

$$\mathcal{L}(P \setminus I) = \{t' \mid \exists t \in \mathcal{L}(P) \textit{ such that } t' = t|_{\Sigma \setminus I}\}$$

That is, the traces accepted by $P \setminus I$ are the projections on $\Sigma \setminus I$ of those accepted by P.

2.4 Scenarios as message sequence charts

Labeled transition systems may capture the behaviors of a single agent class. Scenarios illustrate admissible interactions among multiple agent instances. The scenarios we use in this thesis are specified in a syntactic subset of Message Sequence Charts (MSC) [ITU96], see Fig. 2.1 for an example.

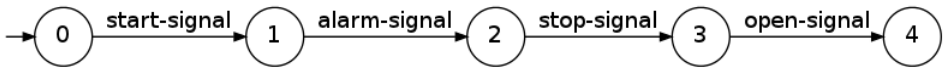


Figure 2.6: LTS capturing the traces of the MSC in Fig. 2.1 from the local perspective of the `Controller` agent.

To keep scenario specifications accessible to end-users, we will consider only a small subset of their features. In its simplest form, a MSC is composed of vertical lines representing timelines associated with agent instances and horizontal arrows representing interactions events among them. According to the previous section, events are synchronously sent and received by interacting agents (we will also use the terms *controlled* and *monitored* events, respectively). We assume that an event label uniquely determines the latter agents.

We consider *positive* scenarios, that are examples of behaviors that the system should exhibit (see Section 2.4.1), and *negative* scenarios that are counterexample behaviors that the system must avoid (see Section 2.4.2). Sections 2.4.3 and 2.4.4 will discuss ways of managing multiple positive and negative scenarios. The consistency of the scenarios and state machines views is discussed in Section 2.4.5.

2.4.1 Positive scenarios

As in [Uch04], MSCs are given a trace semantics. We will consider that a MSC defines a set of traces; the latter are expressed through a LTS. Two kinds of traces are considered: those from the local perspective of a single timeline and those from the global perspective of the complete MSC. We discuss each of these views in turn.

As time in a MSC evolves from top to bottom, the order in which events are sent and received along a particular timeline defines a total order. Therefore, from the perspective of a single agent, a MSC defines one simple trace; such trace is a *maximal* trace in that it includes all events in which the agent participates. This trace and all its prefixes can be captured by a LTS. Given a MSC M and an agent Ag, we will denote such LTS by $M_{\downarrow Ag}$.

For example, the traces defined by the timeline of the `Controller` in the MSC of Fig. 2.1 are precisely captured by the LTS in Fig. 2.6.

When looking at traces from the perspective of the whole MSC, two possibilities might be envisaged.

Total event ordering – One might consider that a MSC defines a simple *maximal* trace where all events appear according to the graphical top-down ordering. In the example of Fig. 2.1, such trace would be

```
<start-signal, start, alarm-pressed, ..., open>
```

A MSC would then define a total order among all events. This leads to a straightforward but limited trace semantics for MSCs.

Partial event ordering – When considering concurrent systems, a partial ordering among events appears more adequate [ITU96, Uch03].

Consider for example the events `start-signal` and `alarm-pressed` at the beginning of the MSC shown in Fig. 2.1. These two events capture unrelated message passing between different agents; therefore, they cannot be considered strictly ordered over time – e.g. the passenger might push the alarm button when the `start-signal` is already sent but before `start` has been propagated; or maybe even before `start-signal`; and so on.

To account for such situations, we need to consider that the traces defined by a MSC are *linearizations* of the partial order among MSC events [Alu00]. In other words, linearizations capture all possible sequences of events that respect the total ordering defined by the timelines. We do not formalize the structure of Message Sequence Charts and their linearizations here, and refer to [Uch03] for such mathematical characterization.

Let M denote a MSC. The set of traces it defines from the local and global perspectives are related as follows:

$$\mathcal{L}_{total}(M) \subseteq \mathcal{L}_{partial}(M) \tag{2.3}$$

$$\mathcal{L}_{partial}(M) = \mathcal{L}(M_{\downarrow Ag_1} \parallel \ldots \parallel M_{\downarrow Ag_n}) \tag{2.4}$$

- Relation (2.3) states that the set of traces under a partial ordering includes those under a total ordering. Clearly, the model with partial ordering is more general than the one with total ordering. This means that everything that is true for the former is also true for the latter. Unless stated otherwise, we will therefore assume the general framework with *partial ordering*. In particular, we will no longer make the *partial* and *total* subscripting explicit when stating other language relations in the following sections.

- Relation (2.4) provides a way of computing all linearizations of a MSC as an acyclic transition system through LTS composition. Such LTS is illustrated in Fig. 2.7 for the MSC in Fig.2.1. As shown there, the latter accepts six different linearizations due to the possible interleavings of its first four events. By construction, such LTS has only one initial state (the leftmost one) and only one terminating state (the rightmost one).

Note that the number of linearizations of a MSC is exponential in the number of events in the worst case. This is representative of distributed

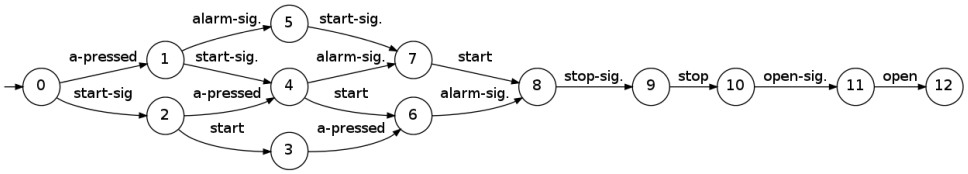


Figure 2.7: LTS capturing all event linearizations of the MSC in Fig. 2.1. Here, `alarm-pressed` is abbreviated as `a-pressed` and `sig.` stands for `signal`.

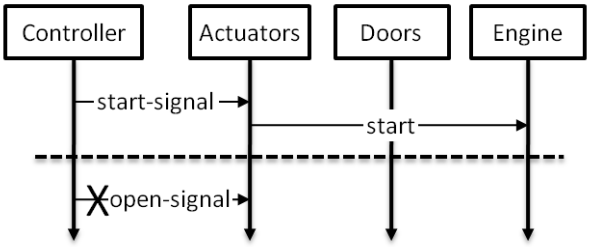


Figure 2.8: A negative scenario illustrating that the controller may not open doors after having started.

systems where agents generally behave asynchronously. Linearizations capture all possible interleavings of events in exactly the same way as LTS composition in Section 2.3.2. In the worst case where agents do not synchronize at all, the number of interleavings is exponential. In practice, as they specifically illustrate agent interactions, MSCs often have only a few linearizations.

2.4.2 Negative scenarios

Positive MSCs capture examples of behavior that the system is expected to exhibit. In addition, it is often convenient to specify examples of behavior that the system may *not* exhibit. Proscribed behaviors are illustrated through negative MSCs [Uch02, Uch04]. A negative MSC is a scenario whose last event is prohibited, as depicted by a crossed arrow below a dashed line. Fig. 2.8 shows a negative scenario where the `Controller` may not open doors immediately after having started the train.

More precisely, a negative MSC is a pair (P, e) where P is a positive MSC and e is a single event label. The positive scenario prefix P is called the *precondition* and e the *prohibited event*. The intuitive semantics is that, once the precondition has occurred from the system's initial state, e may not be the very *next* event in the system.

We now make the trace semantics of negative MSCs precise. First, the

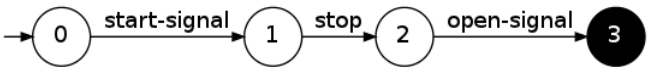


Figure 2.9: Negative traces for the negative MSC of Fig. 2.8, captured with a LTS extended with an error state

precondition of a negative MSC $N = (P, e)$ is a positive MSC; it therefore defines a set of positive traces $\mathcal{L}^+$:

$$\mathcal{L}^+(N) = \mathcal{L}(P) \tag{2.5}$$

In accordance with the previous section, we implicitly assume the general framework with partial ordering here.

A negative MSC $N = (P, e)$ defines a set of negative traces $\mathcal{L}^-$:

$$\mathcal{L}^-(N) = \{\ we \mid w \in mt(\mathcal{L}(P))\ \} \tag{2.6}$$

where $mt(\mathcal{L})$ was seen to denote the set of maximal traces of the language $\mathcal{L}$ (see Section 2.3.1).

Negative traces are thus maximal traces of the precondition concatenated with the label of the proscribed event. Note that the precondition must occur completely for the prohibited event to be taken into account. In other words, partial orderings between the prohibited event and those in the precondition are not considered. This is the intended meaning of the dashed line separating them [Uch04].

Note that the negative language of a negative MSC cannot be captured through a pure LTS. This is because $\mathcal{L}^-(N)$ is not prefix-closed (see Section 2.3.1). Negative scenarios are sometimes captured by a LTS extended with an error state, see Fig. 2.9 where the error state is depicted in black. Alternatively, we may use a standard automaton that makes a distinction between accepting and non-accepting states.

2.4.3 Scenario collections

Systems are generally illustrated through multiple positive and negative scenarios. We will therefore consider scenario collections $Sc = (S^+, S^-)$ where S^+ and S^- are finite, possibly empty, sets of positive MSCs and negative MSCs, respectively.

A common assumption when using scenario collections is that *all scenarios, positive or negative, start in the same system state.* In section 2.4.4 we will show how this assumption can be removed.

Under this assumption, the trace semantics of a scenario collection is captured via the union operator on languages. A scenario collection $Sc =$

(S^+, S^-) defines a positive and a negative language. For the former, the definition below takes into account the fact that negative scenarios define positive traces in addition to negative ones:

$$\mathcal{L}^+(Sc) = \bigcup_{P \in S^+} \mathcal{L}(P) \quad \cup \quad \bigcup_{N \in S^-} \mathcal{L}^+(N)$$
$$\mathcal{L}^-(Sc) = \bigcup_{N \in S^-} \mathcal{L}^-(N)$$

2.4.4 Flowcharting scenarios in high-level message sequence charts

A high-level Message Sequence Chart (hMSC) is a directed graph where each node refers to a MSC or a finer grained hMSC [ITU96]. The MSCs are then called *basic* MSCs, bMSCs for short. The outgoing edges from a node capture possible continuations, allowing the user to introduce sequences, alternatives and loops; to reuse small MSC fragments; and so on. A hMSC has an initial starting point that indicates the initial system state. Fig. 2.10 shows a hMSC for our running example.

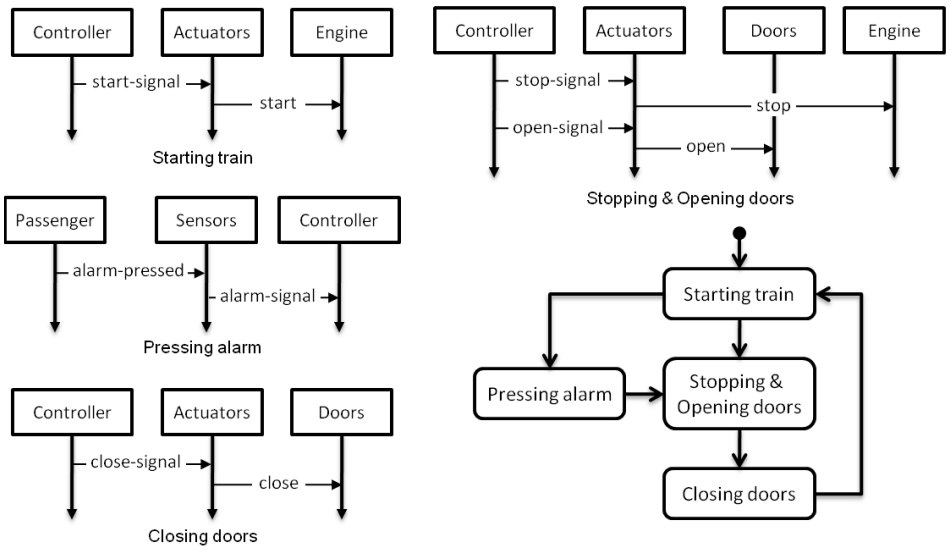


Figure 2.10: A high-level Message Sequence Chart for the train system.

The two following sections define the trace semantics of hMSCs. We first restrict to hMSCs composed of bMSCs only before taking finer-grained hMSCs into account.

hMSCs composed from bMSCs only

A trace semantics for hMSCs provides an answer to the question: *what traces are captured by a hMSC?* This apparently simple question has a fairly complex answer. The reasons are manifold; we slightly extend the characterization provided in [Uch04].

Bounding hMSCs – As explained in [Hen00], some hMSCs do not define regular languages. In that case, they have sets of traces that cannot be captured by a LTS. Our framework must therefore be restricted to regular hMSCs. Under a total ordering of events inside bMSCs, hMSCs are regular. Under a partial ordering, a sufficient condition for a hMSC to be regular is that it does not contain a cycle in which two disjoint sets of agents interact independently of each other. We will make the assumption of *bounded* hMSCs in the sequel.

Composing bMSCs – In addition to the partial or total ordering of events in bMSCs, two possibilities arise as to how a system evolves from a bMSC to another inside a hMSC. The first one, called *strong sequential composition*, assumes that all agents wait until all events of a bMSC have occurred before moving to the next one. This means that there is an implicit synchronization scheme used by the agents to know when a scenario has been completed. The other one, called *weak sequential composition*, allows an agent to move from a bMSC to another one without having to synchronize with the other agents.

In view of the independence between the assumptions of partial/total event ordering and weak/strong sequential composition, four combinations actually exist. To keep things simple enough and avoid bizarre concurrency models[2], we do not consider partial (resp. total) ordering with strong (resp. weak) sequential composition. In the sequel, therefore, strong (resp. weak) sequential composition will entail total (resp. partial) ordering of MSC events. In the sequel, we will refer to this as the *stable composition* hypothesis.

Trace semantics – Under the assumption of bounded hMSC with stable composition, the trace analysis for hMSCs is very similar to MSCs. It leads to language relations similar to the ones given for the latter in Section 2.4.1 – see (2.3) and (2.4) in particular:

$$\mathcal{L}_{strong}(H) \subseteq \mathcal{L}_{weak}(H) \tag{2.7}$$

$$\mathcal{L}_{weak}(H) \subseteq \mathcal{L}(H_{\downarrow Ag_1} \parallel \ldots \parallel H_{\downarrow Ag_n}) \tag{2.8}$$

[2]In particular, such models are very sensitive to the decomposition of a hMSC into bMSCs. For example splitting a bMSC node into smaller bMSCs might change the set of accepted traces of the hMSC.

The sets of traces $\mathcal{L}_{strong}(H)$ and $\mathcal{L}_{weak}(H)$ result from the scenarios that can be "produced" by the hMSC under strong and weak sequential composition, respectively. Such "production" simply results from the concatenation of bMSCs along paths admissible by the hMSC.

For example, concatenating the bMSCs *Starting train*, *Pressing alarm* and *Stopping & Opening the doors* in the hMSC of Fig. 2.10 leads to the MSC of Fig. 2.1.

Such MSC M defines the set of traces $\mathcal{L}_{total}(M)$ and $\mathcal{L}_{partial}(M)$, see Section 2.4.1. The trace semantics of a hMSC H is defined in terms of all possible MSCs that it produces:

$$\mathcal{L}_{strong}(H) = \bigcup_{M \in H} \mathcal{L}_{total}(M)$$
$$\mathcal{L}_{weak}(H) = \bigcup_{M \in H} \mathcal{L}_{partial}(M)$$

where $M \in H$ means "the MSC M can be produced by a path in H". The language $\mathcal{L}_{weak}$ corresponds to the notion of *trace model of a hMSC* in [Uch04].

Another way of defining the set of traces of a hMSC consists in considering the local perspective of the agent timelines. This is similar to what has been done for MSCs in relation (2.4) and leads to the right term in (2.8). There, hMSC traces are computed as the composition of local agent traces $H_{\downarrow Ag_i}$ (see below). The composition of such LTS for each agent corresponds to the notion of *minimal architecture model* in [Uch04]. We denote it by $\mathcal{L}_{arch}(H)$:

$$\mathcal{L}_{arch}(H) = \mathcal{L}(H_{\downarrow Ag_1} \parallel \ldots \parallel H_{\downarrow Ag_n}) \tag{2.9}$$

To sum up, defining the trace semantics of a hMSC H leads to considering three sets of traces, namely $\mathcal{L}_{strong}(H)$, $\mathcal{L}_{weak}(H)$ and $\mathcal{L}_{arch}(H)$. The first two sets consider the global perspective of the hMSC whereas the third set considers the local perspective of agent timelines. Moreover, $\mathcal{L}_{strong}(H)$ assumes a strong sequential composition of bMSC nodes with total event ordering whereas $\mathcal{L}_{weak}(H)$ and $\mathcal{L}_{arch}(H)$ both assume weak sequential composition and partial ordering. These three languages are related as follows:

$$\mathcal{L}_{strong}(H) \subseteq \mathcal{L}_{weak}(H) \subseteq \mathcal{L}_{arch}(H)$$

In practice, compositional algorithms allow capturing these three languages through LTS:

- An algorithm for synthesizing a LTS for $\mathcal{L}_{arch}(H)$ may be found in [Uch03]. For a given agent Ag_i, the LTS $H_{\downarrow Ag_i}$ is synthesized by connecting the LTSs $M_{\downarrow Ag_i}$ corresponding to each bMSC M with τ transitions, according to their possible continuations given by hMSC edges.

  This construction is illustrated in Fig. 2.11 for the hMSC in Fig. 2.10 and the `Controller` agent. The resulting LTS can be further simplified by removing τ transitions and minimizing the result.

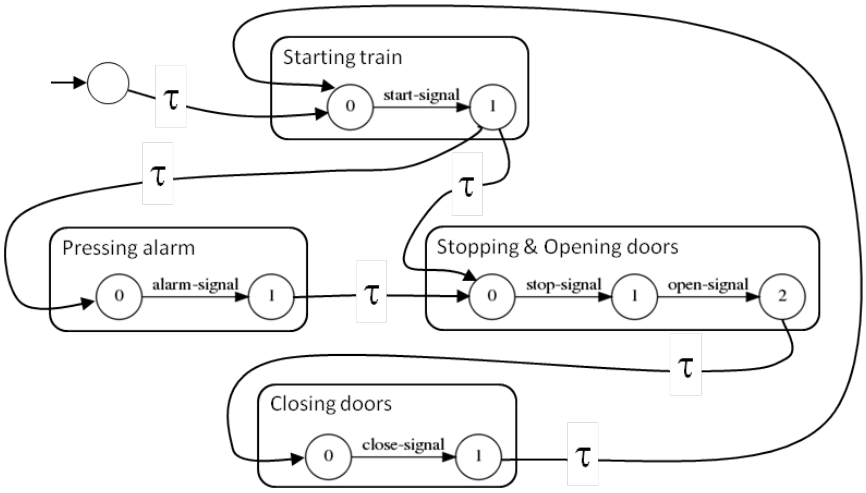


Figure 2.11: Synthesis of the *Controller* LTS from the hMSC of Fig. 2.10.

- Thanks to the simplicity of the model with strong sequential composition, $\mathcal{L}_{strong}(H)$ can be captured in a similar way. The linear event trace defined by each MSC is used instead of $M_{\downarrow Ag_i}$ in the algorithm above.
- The synthesis of a LTS for $\mathcal{L}_{weak}(H)$ is more complex because the sequencing and synchronization of bMSCs must be constrained to fit the semantics. A synthesis algorithm can be found in [Uch04].

An important difference should be noted among the relations between MSC languages and those on hMSC languages. For similarity with hMSCs, let $\mathcal{L}_{arch}(M)$ denote an architecture model for MSCs:

$$\mathcal{L}_{arch}(M) = \mathcal{L}(M_{\downarrow Ag_1} \parallel \ldots \parallel M_{\downarrow Ag_n})$$

For a MSC M and a hMSC H the language relations are, respectively:

$$\mathcal{L}_{total}(M) \subseteq \mathcal{L}_{partial}(M) = \mathcal{L}_{arch}(M) \tag{2.10}$$

$$\mathcal{L}_{strong}(H) \subseteq \mathcal{L}_{weak}(H) \subseteq \mathcal{L}_{arch}(H) \tag{2.11}$$

Note the equality between $\mathcal{L}_{partial}(M)$ and $\mathcal{L}_{arch}(M)$ in (2.10) and the set inclusion between $\mathcal{L}_{weak}(H)$ and $\mathcal{L}_{arch}(H)$ in (2.11).

The traces in $\mathcal{L}_{arch}(H) \setminus \mathcal{L}_{weak}(H)$ capture the set of *implied* scenarios of a hMSC specification [Alu00, Uch04]. Implied scenarios occur when a system is designed globally while implemented component-wise. They capture traces that follow different paths in the hMSC when projected on individual agent timelines. Implied scenarios will be mostly ignored in the thesis; we will however discuss their impact on our inductive technique in Section 4.6.

hMSCs composed from finer-grained hMSCs

A hMSC node may refer to a finer-grained hMSC instead of a bMSC. To take this into account in the trace semantics given previously, we must decide how a sub-hMSC is to be "connected" with its father. To keep a consistent framework in terms of synchronization hypotheses, we require such sub-hMSC to have, in addition to its initial state, a terminating state to which at least one node is connected. For simplicity, we forbid nodes with no outgoing transition.

Under those assumptions, it is fairly easy to unfold a compound hMSC into another where all refined nodes are basic MSCs. The trace semantics remains unchanged and is defined in terms of the latter "flat" hMSC.

In practice, this flat hMSC must not be explicitly constructed when the LTS for $\mathcal{L}_{arch}$ is synthesized. Under our assumptions, the LTSs capturing the traces of finer-grained hMSCs have only one initial state and only one terminating state. Therefore, they can connected with τ transitions in the same way as the LTSs $M_{\downarrow Ag_i}$ in the synthesis algorithm of previous section. A similar argument applies for $\mathcal{L}_{strong}$.

2.4.5 Consistency between the scenario and state machine views

The trace semantics of scenarios can be explicitly related to the trace semantics of agent and system state machines. This section defines consistency rules between those two views, thereby explaining the similarity between relations (2.2), (2.4) and (2.8).

Positive MSCs

Consider a system composed of n agents whose behavior is modeled by $S = (Ag_1 \parallel \ldots \parallel Ag_n)$. Let M denote a positive MSC illustrating interactions among them. M and S are said to be *consistent* if the following conditions hold:

Structural consistency – This condition requires M and S to agree on the set of agents and their respective interfaces. The MSC may actually illustrate interactions among a proper subset of system agents. However, labels along a timeline must be a subset of the alphabet of the corresponding agent.

Consistent agent view – This condition states that the traces defined by a timeline in the MSC must be traces accepted in the LTS modeling the behavior of the corresponding agent. For each agent Ag_i, we must have:

$$\mathcal{L}(M_{\downarrow Ag_i}) \subseteq \mathcal{L}(Ag_i). \tag{2.12}$$

Consistent system view – This third condition states that all linearizations of the MSC must be accepted traces in the LTS modeling the behavior of the composed system:

$$\mathcal{L}(M) \subseteq \mathcal{L}(S), \tag{2.13}$$

where $\mathcal{L}(M) = \mathcal{L}(M_{\downarrow Ag_1} \parallel \ldots \parallel M_{\downarrow Ag_n})$

Negative MSCs

A negative MSC $N = (P, e)$ and a system $S = (Ag_1 \parallel \ldots \parallel Ag_n)$ are said to be consistent if the following conditions hold:

- N and S are *structurally* consistent, in a similar sense to what has been said for positive MSCs,
- the precondition P and S are consistent; this requires conditions (2.12) and (2.13) to hold for P,
- the system may not exhibit any trace captured by the negative MSC, that is,

  $$\mathcal{L}^{-}(N) \cap \mathcal{L}(S) = \emptyset \ , \tag{2.14}$$

  which provides the negative counterpart of (2.13).

Scenario collections

A scenario collection $Sc = (S^{+}, S^{-})$ and a system S are said to be consistent if and only if each positive and each negative MSC of the collection is itself consistent with S.

Conditions (2.13) and (2.14) can be formulated in terms of the positive and negative languages of the scenario collection as follows:

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(S) \tag{2.15}$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(S) = \emptyset \tag{2.16}$$

Note that a scenario collection may only be consistent if all its scenarios, positive or negative, start in the same system state.

A set S^+ of positive scenarios and a set S^- of negative scenarios are consistent with each other if there exists a system which is consistent with them taken as a collection $Sc = (S^+, S^-)$.

A necessary condition for a scenario collection $Sc = (S^+, S^-)$ to be consistent is the disjointness of positive and negative traces:

$$\mathcal{L}^+(Sc) \cap \mathcal{L}^-(Sc) = \emptyset \tag{2.17}$$

This condition is not sufficient, however, as structural consistency is not guaranteed.

High-level MSCs

A hMSC H and a system $S = (Ag_1 \parallel \ldots \parallel Ag_n)$ are said to be consistent if the following conditions hold:

- H and S are *structurally* consistent,
- $\mathcal{L}(H_{\downarrow Ag_i}) \subseteq \mathcal{L}(Ag_i)$ for each agent Ag_i,
- $\mathcal{L}_{arch}(H) \subseteq \mathcal{L}(S)$,

where $\mathcal{L}_{arch}(H) = \mathcal{L}(H_{\downarrow Ag_1} \parallel \ldots \parallel H_{\downarrow Ag_n})$

These conditions are the counterpart of those given previously for MSCs. They have a similar interpretation, *mutatis mutandis*.

It is sometimes convenient to distinguish between a hMSC that describes all behaviors of a system and one that only illustrates a subset of them. This leads to the notion of hMSC completeness.

A hMSC H is *complete* for a system $S = (Ag_1 \parallel \ldots \parallel Ag_n)$ if it is consistent with it and defines the same language, that is, if the following condition holds:

$$\mathcal{L}_{arch}(H) = \mathcal{L}(S) \tag{2.18}$$

2.5 State-based assertions on 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*" [Mil02].

Fluents will allow us to integrate event-based and state-based specification styles within a simple framework [Gia03].

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 definitions is the following [Gia03]:

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

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

$$\text{fluent } Moving = <\{start\}, \{stop\}> \text{ initially } false$$
$$\text{fluent } DoorsClosed = <\{close\}, \{open\}> \text{ initially } true$$

2.5.1 Fluent values along single traces

In terms of our trace semantics, a fluent Fl will be *true* after a finite trace s if and only if one of the following conditions hold [Gia03]:

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

As sets of initiating and terminating events are disjoint, the value of a fluent after a given trace is deterministic.

For example, the fluent *Moving* is *true* after the trace `<start stop start>`, but not after the empty trace λ nor after `<start stop>`.

As prefixes of traces are also traces, we may also think in terms of fluent values *along* traces. This is illustrated in Fig. 2.12 where the values of the two fluents introduced above are shown in the states of a LTS capturing a typical event trace for the train system.

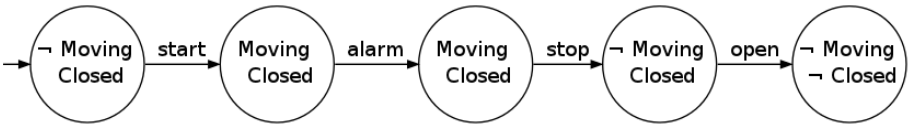


Figure 2.12: LTS annotated with fluent values along a single trace (`Closed` is an abbreviate for `DoorsClosed`).

As the example suggests, for a set of fluents Φ, an event trace yields an interpretation over 2^{Φ}, that is, an assignment of a Boolean value to each fluent in Φ. We will call it *fluent value assignment* throughout the thesis.

For example, the maximal trace `<start alarm stop open>` in Fig. 2.12 yields the following fluent value assignment:

$$\{Moving = false,\ DoorsClosed = false\}$$

2.5.2 Fluent values along multiple traces

We can similarly annotate the states of any LTS. The generalization consists in considering that a LTS state may be reached by a set of traces instead of a single one. Annotations therefore become *sets* of fluent assignments, that is elements of the powerset $\mathcal{P}(2^{\Phi})$ over 2^{Φ}.

In particular, a state might be reached by a trace yielding a fluent *true*, while another trace reaching it would yield the same fluent *false*. Note that even in presence of a possibly infinite number of traces, the set of all possible annotations is itself finite.

$\mathcal{P}(2^{\Phi})$ actually coincides with the set of propositional formulas over fluents. In fact, it appears convenient to consider state annotations as such formulas, where *false* then corresponds to an empty set of fluent assignments (i.e. an unreachable state) and *true* corresponds to the set of all possible assignments. Such state annotations will be called *state invariants*; they encode assertions that always hold when the LTS state is visited.

Fig. 2.13 shows an example of LTS annotated with state invariants.

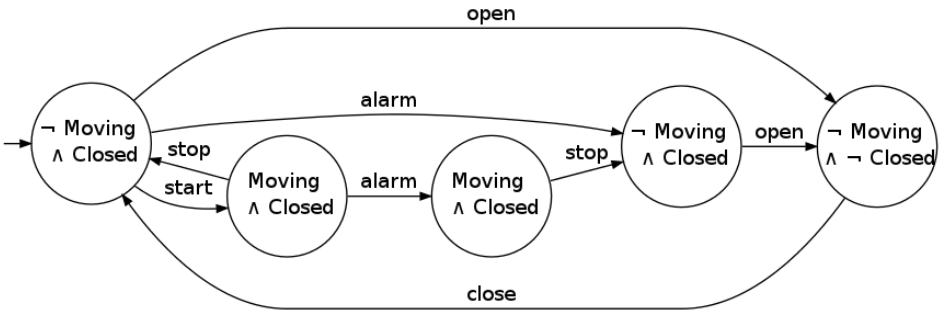


Figure 2.13: Fluent values along multiple traces (`Closed` is an abbreviate for `DoorsClosed`).

A fixpoint algorithm for decorating LTS with fluent value assignments appeared in [Dam05]. This algorithm was generalized for generating weakest state invariants in [Dam09, Dam11]. Very roughly, it consists in annotating the LTS initial state with an initial invariant according to fluent initial values; invariants are then propagated along LTS transitions, according to fluent definitions, and accumulated in states through Boolean disjunction, until a fixpoint is reached.

Binary decision diagrams [Bry86] can be used for concisely encoding and efficiently manipulating Boolean formulas (see Chapter 7). This algorithm has been further generalized [Dam11] into a fully generic version whose instantiations support other kinds of decorations beside state invariants.

2.5.3 Integrating fluents in multi-view models

Fluents provide a general mechanism for integrating event-based and state-based specifications. We will use them for integrating agents, their state machines, and scenarios. The next section extends such integration by introducing fluent-based goals and domain properties.

Our use of fluents will be restricted to the ones that are monitorable and controllable by the agents forming the system [Let02].

A fluent is *monitorable* by an agent if all its initiating and terminating events can be either sent or received by this agent. It is *controllable* if all initiating and terminating events can be sent by the agent.

Our restriction to fluents that are monitorable or controllable by system agents is motivated by the need for specifications to be realizable by the agents involved in them [Let02].

2.6 Declarative goals and domain properties

A *goal* is a prescriptive statement of intent whose satisfaction requires the collaboration of agents forming the system. Unlike goals, *domain properties* are descriptive statement about the environment – such as physical laws, organizational rules, etc. Goals are structured in AND/OR refinement graphs that show how they contribute to each other [van00b].

Section 2.6.1 formalizes goals and domain properties in fluent linear temporal logic (FLTL). Their integration with state machines and scenarios is discussed in Section 2.6.2. Section 2.6.3 then presents how all system traces satisfying (resp. violating) a goal can be specified through a property (resp. a tester) automaton, provided that the goal captures a safety property.

2.6.1 Properties as FLTL assertions

Goals and domain properties can be formalized in Linear Temporal Logic (LTL). Admissible system histories are thereby specified in a declarative and implicit way [van09]. We will focus on propositional LTL here. In addition to the usual propositional constructs, LTL provides connectives for temporal referencing [Man92]:

- $\circ$ (at the next smallest time unit),
- $\diamond$ (some time in the future),
- $\Box$ (always in the future),
- $\rightarrow$ (implies in the current state),
- $\Rightarrow$ (always implies),
- $\mathcal{U}$ (always in the future until),
- $\mathcal{W}$ (always in the future unless)

A system history is commonly seen in LTL as a temporal sequence of system states. Atomic LTL propositions then refer to state formulas (as, e.g. in the SPIN model-checker [Hol97]). In our event-based setting, a system history will be seen as a trace, that is, a temporal sequence of events.

To integrate the state-based and event-based paradigms, we will use a flavor of LTL known as Fluent Linear Temporal Logic (FLTL), where atomic propositions are fluents [Gia03]. FLTL proves convenient for specifying state-based temporal logic properties over the event-based operational model given by scenarios and state machines.

For example, the safety goal "*Doors shall remain closed while the train is moving*" of our running example can be formalized in terms of the $Moving$ and $DoorsClosed$ fluents defined in the previous section:

$$\texttt{Maintain[DoorsClosed While Moving]} = \Box(Moving \rightarrow DoorsClosed)$$

Properties formalized in temporal logic are commonly classified as *liveness* or *safety* properties [Alp86, Lam94]. Liveness refers to "*something good will eventually happen*" whereas safety refers to "*something bad may never happen*". The thesis will focus on safety properties for the following reasons:

- Reasoning about liveness requires considering the acceptance of infinite execution traces; this is outside the expressiveness of our LTS characterization and regular languages in general. In contrast, if "something bad" happens it must do so after a finite sequence of events, and is irremediable [Alp86, Gia99].
- The thesis focusses on requirements engineering models. To be realizable by agents, the properties assigned to them must be bounded as *maintain* or *bounded achieve* specifications [Let02]; these correspond to safety properties.

2.6.2 Consistency between the behavioral and intentional views

Consider a safety property G and a system $S = (Ag_1 \parallel \ldots \parallel Ag_n)$. Let $\mathcal{L}^-(G)$ denote the set of event traces violating the property and $\mathcal{L}^+(G)$ those satisfying it. We discuss how these languages can be captured through automata in the following section.

The system S and the property G are consistent if and only if the following condition holds:

$$\mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) \cap \mathcal{L}^-(G) = \emptyset \tag{2.19}$$

The above relation requires the system to exclude any trace violating the property. This relation is the one commonly used in model checking, provided that $\mathcal{L}^-(G)$ actually amounts to $\mathcal{L}^+(\neg G)$ [Cla89]. In other words, $\mathcal{L}^-(G)$ actually captures the set of traces satisfying the negation of the safety property.

We will also consider the following consistency rules between goals and scenarios:

- Behaviors illustrated in positive scenarios may not violate safety properties. A more formal expression of this depends on the concrete form

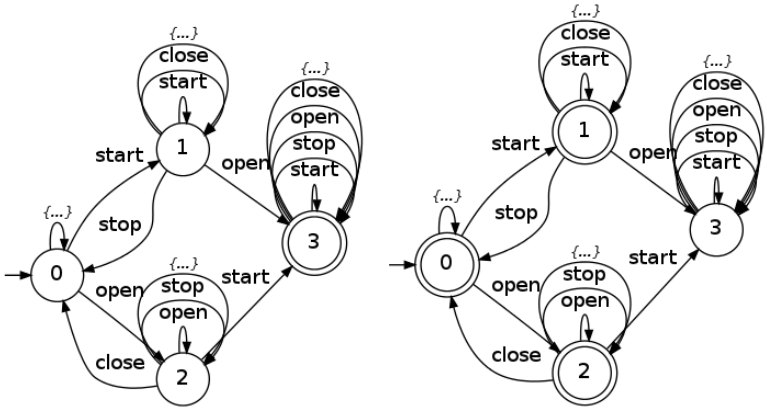


Figure 2.14: Tester ($\mathcal{L}^-$ at left) and property ($\mathcal{L}^+$ at right) automata for `Maintain[DoorsClosed While Moving]`, as complements of each other.

of scenario specification (scenario collection or hMSC) and on hypotheses about event ordering and weak or strong bMSC synchronization. It is similar to relation (2.19), *mutatis mutandis*.

- A negative scenario N *illustrates* the violation of a safety property, say G, if the following condition holds:

$$\mathcal{L}^-(N) \subseteq \mathcal{L}^-(G) \tag{2.20}$$

that is, if the negative traces it defines are among those prescribed by the safety property.

2.6.3 Property and tester automata

For a safety property G, the sets of traces $\mathcal{L}^+(G)$ and $\mathcal{L}^-(G)$ are complement of each other, that is,

$$\mathcal{L}^+(G) = \Sigma^* \setminus \mathcal{L}^-(G) \tag{2.21}$$

Both languages can be specified through standard automata. The one for $\mathcal{L}^+(G)$ will be called *property* automaton [Let05] whereas the one for $\mathcal{L}^-(G)$ will be called *tester* automaton [Gia03]. Unlike those references, our characterization here makes use of standard automata instead of LTS.

An example of tester and property automata is given in Fig. 2.14 for the safety goal requiring the doors to remain closed while the train is moving.

A procedure for computing tester automata in the case of safety properties is given in [Gia03]. It consists in building a Büchi automaton that

recognizes all infinite event-based traces violating the state-based FLTL property. Given a pure safety property, a canonical form for this automaton can been found; the latter is deterministic, has a complete transition function, and a unique sink accepting state reached by all traces violating the property [Gia03].

As tester and property automata are complements of each other, it suffices to flip accepting and non-accepting states of one automaton to obtain the other [Hop79]. A necessary condition, guaranteed by [Gia03], is that they have complete transition functions.

A few remarks are in order here.

- In the LTSA toolset [MK99] and related literature, notably [Gia03], a tester LTS with an error state is used instead of the tester automaton presented here. Similarly, the property LTS is sometimes built by removing the error state of the tester LTS [Let08].

  Our automata variants are better suited to our context in view of the formalization of inter-model consistency rules expressed in terms of set-theoretic operators on languages.

- The tester and property automata, together with the procedure for computing them, are always defined "up to a given alphabet" or "under the assumption of a given agent or system". This is the intended meaning of the transitions labeled "{...}" in Fig. 2.14. The reason is that some temporal properties, in particular those referring to the FLTL *next* operator, are not closed under stuttering [Lam94]. This means that their satisfaction may be affected by the insertion or removal of unobservable events. Which events are relevant may depend on additional hypotheses about how goals relate to agent behaviors – e.g, whether they are under the responsibility of a single agent or are properties to be met by the global system. Fixing those "relevant events" by filling the "{...}" placeholder is required for meeting the temporal logic semantics when composing tester and property automata.

  For simplicity in the thesis, we will consider that safety properties must be met by the global system. Therefore, the relevant events are the alphabet of the whole system, say Σ. A placeholder then must be replaced by a transition for each event in Σ except those already labeling an outgoing transition from the same state.

2.7 Process models as guarded high-level message sequence charts

Process models capture tasks performed by agents together with their control flow. The aim here is to model work processes for effective software support [Dvt05]. A process lifecycle typically includes *process modeling* [Obj04, Obj08, Cla08, Dam09]; *process enactment* [Man02, Buh05, Sa06]; *process monitoring* [Mue00]; and *process restructuring* based on monitored data.

Process models considered in the thesis will be formalized as guarded high-level message sequence charts (g-hMSC), an extension of hMSCs with guards on fluents [Dam09]. Guarded hMSCs capture *decision-based processes* where decisions relying on the state of the process environment regulate the nature of the subsequent tasks and their composition.

An example of g-hMSC is given in Fig. 2.2 for the meeting scheduling case study. Such processes are frequently found in medical therapies, for example.

A task is a unit of work to be performed by collaboration of agents involved in the process.

Task nodes in a g-hMSC may be refined either by a MSC or by a finer-grained g-hMSC. For example, the `InitiateMeeting` task in Fig. 2.2 is refined into a MSC illustrating interactions between the initiator, the scheduler and participants.

A g-hMSC defines a strong sequential composition of task nodes; a total ordering of all events also applies inside MSCs (see Section 2.4.4). This assumes an implicit synchronization scheme used by the agents to know when tasks are starting and terminating. Automated process enactment typically provides such support through agent worklists, reminders sent to them, and the like.

Such synchronization is kept implicit in the process model. However, for reasoning about timing of tasks and manipulating the traces that a g-hMSC accepts (see Chapter 3), it proves convenient to associate with a task special events for denoting its start and end. For a task T such events will be denoted by T_{start} and T_{end}, respectively. They correspond to *broadcasting* interactions, which implies that all agents are supposed to monitor them.

A decision node states specific conditions for the tasks along outgoing branches to be performed. Each outgoing branch is labeled by a Boolean expression on fluents, called *guard*. A guard must be evaluated to true for the corresponding branch to be followed. It captures

a condition under which the subsequent task flow *may* and/or *may not* be performed. g-hMSC do not provide abstractions for capturing an obligation to perform a task, that is, when the latter *must* be performed.

For example, the guards in Fig. 2.2 refer to the three fluents: *second_cycle*, *date_conflict* and *resolve_by_weakening*. The former is defined as below; the two others are to be defined in terms of events introduced in refinements of other tasks.

$$\text{fluent } second\_cycle = \\ <\{ExtendDateRange_{end}, WeakenConstraints_{end}\}, \\ \{InitiateMeeting_{end}\}>$$

Instead of initial values for fluents, a g-hMSC is given an *initial condition* that constrains the acceptable initial fluent values. This allows processes to be modelled where different instances can start in different states; initial values for fluents can then be defined at instance level rather than at class level. The initial condition will be denoted by C_0. In our example, it might be defined as follows:

$$C_0 = \neg second\_cycle \wedge \neg date\_conflict \wedge \neg resolve\_by\_weakening$$

The trace semantics of guarded hMSCs will be defined in Chapter 3 together with synthesis techniques to derive them as state machines for automated reasoning.

Summary

This chapter described the multi-view formal framework used for system modeling in the thesis. This framework is composed of scenarios, state machines, goals and process models. All models are semantically grounded in trace theory. Fluents are used for capturing agent and process state variables; they allow integrating event-based and state-based models. Formal rules have been defined that capture consistency conditions between the different views of the system modeled.

Chapter 3

Derivation of State Machine Models from Process Models

This chapter shows how LTS state machines can be synthesized from g-hMSC process models. Section 3.1 motivates such derivation and outlines the approach. Section 3.2 introduces guarded labeled transition systems (g-LTS) as an intermediate formalism between guarded hMSCs and LTS state machines. Algorithms for transforming process models into guarded and then guard-free transition systems are detailed in Section 3.3.

3.1 Motivation and approach

Building adequate, complete and consistent process models is not necessarily an easy task. Flawed models can have dreadful consequences in safety-critical areas. The recent usage of process modeling to improve medical safety provides a good example of such area [Cla08, Gra09, Dam11]; models there should be as error-free as possible. Techniques should therefore be available to support model construction and analysis, in particular for systematically detecting and fixing severe flaws.

A typical analysis technique coming to mind is *model checking* [Cla89]. Model-checking would allow us to verify that a given safety property is satisfied in all process instances. Other kinds of analysis on process models are discussed in [Dam11], including checking the completeness, disjointness and satisfiability of guards at decision points; verifying that timing requirements are met; detecting inadequate process decisions; and so on. This kind of check should be performed ahead of process enactment.

Verification techniques require a formal semantics for process models. As our guarded hMSCs are an extension of hMSCs, a trace semantics appears natural. This choice also fits the intuition: an event trace can roughly be

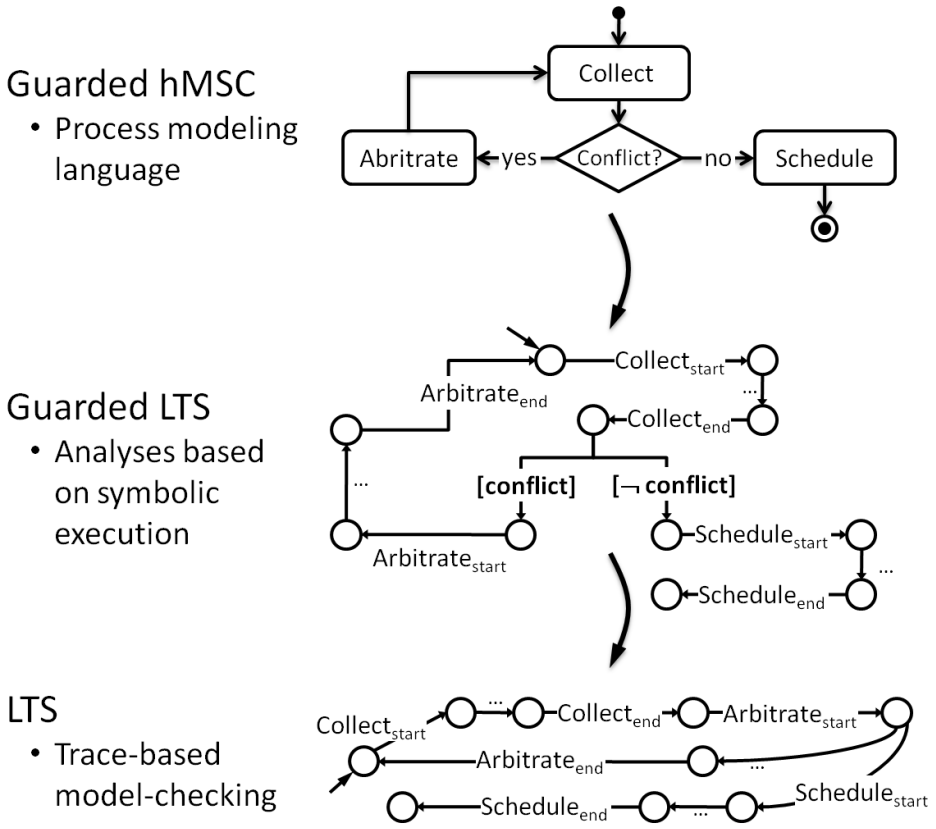


Figure 3.1: Guarded LTS as an intermediate level between g-hMSC and LTS.

rephrased in terms of temporal occurrences of tasks. Moreover, a trace semantics for g-hMSCs enables the reuse of LTS-based tools and techniques [MK99, Gia03].

We will only consider the traces of a process globally, that is, as sequences of events seen by an external observer. In other words, in spite of a multi-agent view through message sequence charts, we will not consider agent state machines in isolation. The aim of process modeling is *not* to specify valid executions of the agents themselves, but to capture the control flow of their tasks.

As a consequence, we will assume a strong sequential composition of g-hMSC nodes and a total ordering of all events inside MSCs (see Section 2.4.4). In other words, we will assume that agents synchronize with each other when tasks are started and completed.

To delimit task boundaries in event traces and reason about execution time, each task T will be associated with special events T_{start} and T_{end}.

These events are assumed to be monitored by all agents; thus, they allow explicitly capturing the required synchronization scheme.

Instead of transforming a guarded hMSC into a LTS directly, we will generate an intermediate form called guarded labeled transition system (g-LTS). Roughly, a g-LTS is a transition system with guards or events on transitions. It is a structured form of LTS aimed at avoiding state explosion in their representation. This intermediate form is therefore easier to understand. More importantly, g-LTS support a variety of analyses at a more abstract level (e.g. checking guards against completeness, disjointess and satisfiability or checking the adequacy of decisions [Dam11]). It may also facilitate code generation for process enactment.

Figure 3.1 illustrates the three abstraction levels provided by g-hMSC, g-LTS and LTS, respectively. As just said, the analyses about guards, timing requirements and the like are performed at the g-LTS level. They rely on a generalization of the symbolic execution algorithm used to decorate a LTS with assertions on fluents (see Section 2.5). For details about such analyses at the g-LTS level, see [Dam11]. Trace-based model-checking is performed at the LTS level using slight adaptations of existing tools such as LTSA [MK99] (see Section 7.1).

Our two derivation algorithms for transforming a guarded hMSC to a guarded LTS then to a LTS are detailed in Section 3.3. Section 3.2 first introduces guarded LTS formally; it also provides a few operators for manipulating them.

3.2 Guarded transition systems

This section introduces guarded LTS (g-LTS) as an intermediate formalism between guarded hMSCs and LTS. This formalism provides a convenient milestone on the way from a guarded hMSC to the corresponding LTS, in particular, for determining the set of traces accepted by the guarded hMSC.

Definition 3.1 (Guarded LTS)**.** *A guarded LTS is defined as a structure* $(Q, \Sigma, \Phi, \delta, q_0, C_0)$ *where*

- Q *is a finite set of states,*
- Σ *is a set of event labels,*
- Φ *is a set of fluents defined on* Σ*,*
- δ *is a transition relation* $Q \times L \times Q$*,*
- $q_0 \in Q$ *is the initial state,*

- $C_0 \in \mathcal{P}(2^\Phi)$ *is an initial condition on fluents,*

where the set of transition labels is defined as $L = \Sigma \cup \{\tau\} \cup \mathcal{P}(2^\Phi)$

In a guarded LTS, transitions are labeled either by a guard, an event, or τ. As with LTS, τ-transitions are non-observable from the environment (see Section 2.3). In the sequel, Σ_τ will denote $\Sigma \cup \{\tau\}$.

A guard is a propositional formula over fluents in Φ. The set of such formula is denoted by $\mathcal{P}(2^\Phi)$, as in Section 2.5. Intuitively, the guard must evaluate to true for its transition to be activated. This is made more precise is the next section.

The condition C_0 plays the same role as in guarded hMSCs. It is a Boolean formula that constrains the acceptable initial values of fluents.

3.2.1 Trace semantics

The semantics of g-LTS is defined in terms of event traces involving no guards at all.

Definition 3.2 (g-LTS execution)**.** *An* execution *of a g-LTS* $G = (Q, \Sigma, \Phi, \delta, q_0, C_0)$ *from* q_0 *is a pair* $(Init, <l_0, \ldots, l_n>)$*, where*

- $Init \in 2^\Phi$ *is an initial fluent value assignment mapping every fluent in* Φ *to true or false,*
- $<l_0, \ldots, l_n>$ *is an finite sequence of labels* $l_i \in \Sigma_\tau \cup \mathcal{P}(2^\Phi)$*, each of them being either an event label,* τ*, or a guard.*

In the presence of guards, only specific executions are to be considered valid from the initial state of the g-LTS. This is captured by the following definition.

Definition 3.3 (Valid g-LTS execution)**.** *An execution* $S = (Init, <l_0, \ldots, l_n>)$ *of a g-LTS* $(Q, \Sigma, \Phi, \delta, q_0, C_0)$ *is* valid *from its initial state* q_0 *iff the following* acceptance conditions *are met for every* i $(0 \leqslant i < n)$*:*

trace inclusion:	$\exists q_{i+1} \in Q$ *such that* $(q_i, l_i, q_{i+1}) \in \delta$
admissible start:	$Init \models C_0$
guard satisfaction:	$S_i \models l_i$ *if* $l_i \in \mathcal{P}(2^\Phi)$,

where S_i *is the fluent value assignment after the i-th event in the trace, with* $S_0 = Init$*.*

The trace inclusion condition states that the label sequence must denote an existing path in the automaton.

The admissible start condition states that the initial fluent value assignment must meet the initial condition C_0.

The guard satisfaction condition ensures that all guards are met along the sequence. The fluent value assignment S_i is determined from fluent definitions, as explained in Section 2.5.1.

As with LTS, the trace semantics of g-LTS is defined in terms of accepted *event* traces. Roughly, it consists of all valid executions where τ labels as well as guards are removed. This is precisely captured by the the following definitions.

Definition 3.4 (g-LTS trace)**.** *A g-LTS trace is a pair* $(Init, <e_0, \ldots, e_n>)$ *where*

- $Init \in 2^{\Phi}$ *is an initial fluent value assignment mapping every fluent in* Φ *to true or false,*
- $<e_0, \ldots, e_n>$ *is an finite sequence of event labels, with* $e_i \in \Sigma$*.*

Unlike an execution, a g-LTS trace only contains event labels, without τ labels nor guards. Such trace still includes a fluent assignment *Init* that associates a truth-value to each fluent in the initial state.

Definition 3.5 (Valid g-LTS traces)**.** *The set of g-LTS traces accepted by a g-LTS* $G = (Q, \Sigma, \Phi, \delta, q_0, C_0)$ *is defined as:*

$$\{\ (Init, s) \mid \exists\ (Init, w)\ \textit{a valid execution of } G \textit{ such that } s = w|_{\Sigma}\ \},$$

where $w|_{\Sigma}$ *denotes the projection of* w *over the alphabet* Σ *(see Section 2.3).*

In other words, the set of valid g-LTS traces corresponds to valid executions where τ and guards have been removed.

The traces considered in the definition above are g-LTS traces, that is, they explicitly define an initial fluent assignment *Init*. The set of pure event traces can be defined through existential quantification over such a fluent assignment:

Definition 3.6 (Trace semantics of g-LTS)**.** *The set of pure event traces accepted by a g-LTS G is defined as:*

$$\{\ s \mid \exists\ \textit{Init such that } (Init, s) \textit{ is a valid g-LTS trace for } G\ \}$$

In the definitions above, g-LTS traces are used as a convenient milestone between g-LTS executions and pure event traces. Indeed, the explicit initial fluent assignment $Init$ proves useful for user feedback when model-checking g-hMSC and g-LTS (see Section 7.1).

The next section introduces hiding and composition operators for g-LTS in a way similar to those defined on LTS in Section 2.3.

3.2.2 Hiding and composition in the presence of guards

The hiding operator on g-LTS is very similar to the one on LTS defined in Section 2.3.3.

Definition 3.7 (g-LTS hiding)**.** *The* hiding *of a set of labels L in a g-LTS $G = (Q, \Sigma, \Phi, \delta, q_0, C_0)$ yields the g-LTS*

$$G \setminus L = (Q, \Sigma \setminus L, \Phi, \delta_{hidden}, q_0, C_0),$$

where δ_{hidden} is the smallest relation satisfying the following rules:

$$\frac{G \xrightarrow{l} G'}{G \setminus L \xrightarrow{l} G' \setminus L} \quad l \notin L \qquad \frac{G \xrightarrow{l} G'}{G \setminus L \xrightarrow{\tau} G' \setminus L} \quad l \in L,$$

where $G \xrightarrow{l} G'$ means that the g-LTS $G = (Q, \Sigma, \Phi, \delta, q_0, C_0)$ may transit into a g-LTS $G' = (Q, \Sigma, \Phi, \delta, q_1, C_0)$ through a transition labeled l from its initial state, provided that $(q_0, l, q_1) \in \delta$.

As with LTS, the hiding operator makes a set of labels invisible by replacing them by τ transitions. Note that the definition here allows hiding both guards and events.

The special case where $L = \mathcal{P}(2^{\Phi})$ yields a g-LTS with no guard remaining. We will then consider that the hiding operator actually returns the LTS defined as follows:

$$G \setminus \mathcal{P}(2^{\Phi}) = (Q, \Sigma, \delta_{hidden}, q_0)$$

Definition 3.8 (g-LTS composition)**.** *Let $G = (Q_1, \Sigma_1, \Phi_1, \delta_1, q_1, C_1)$ and $H = (Q_2, \Sigma_2, \Phi_2, \delta_2, q_2, C_2)$ denote two g-LTS. Their* composition *is the following g-LTS:*

$$G \parallel H = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \Phi_1 \cup \Phi_2, \delta, (q_1, q_2), C_1 \wedge C_2),$$

where δ is the smallest relation satisfying the following rules:

$$\frac{G \xrightarrow{l} G'}{G \parallel H \xrightarrow{l} G' \parallel H} \quad l \notin \mathcal{P}(2^{\Phi}),\ l \notin \Sigma_2$$

$$\frac{H \xrightarrow{l} H'}{G \parallel H \xrightarrow{l} G \parallel H'} \quad l \notin \mathcal{P}(2^{\Phi}),\ l \notin \Sigma_1$$

$$\frac{G \xrightarrow{l} G',\ H \xrightarrow{l} H'}{G \parallel H \xrightarrow{l} G' \parallel H'} \quad l \notin \mathcal{P}(2^{\Phi}),\ l \neq \tau$$

$$\frac{G \xrightarrow{g} G',\ H \xrightarrow{h} H'}{G \parallel H \xrightarrow{g \wedge h} G' \parallel H'} \quad g \in \mathcal{P}(2^{\Phi_1}),\ h \in \mathcal{P}(2^{\Phi_2}),\ g \wedge h \not\models false$$

The first three rules are the same as the LTS composition ones (see Section 2.3.2); here, they explicitly exclude guards. The last rule states that the two LTSs must transit on guards together, provided that the guard conjunction is satisfiable. The resulting transition is then labeled with this conjunction.

3.3 From guarded hMSC to pure LTS

This section presents algorithms for explicitly capturing the traces of a g-hMSC through a pure LTS. Section 3.3.1 provides an algorithm for transforming a g-hMSC into a g-LTS. Section 3.3.2 then shows that the set of event traces admitted by a g-LTS can be captured through a LTS.

3.3.1 From guarded hMSC to guarded LTS

A g-hMSC can be transformed into an equivalent g-LTS. The latter abstracts from the agents and captures the set of global behaviors covered by the g-hMSC. The transformation algorithm extends the technique introduced in Section 2.4.4 to synthesize a LTS capturing hMSC traces [Uch04].

Handling nodes – Every g-hMSC node yields a behaviorally equivalent sub-LTS (see dashed rectangles at bottom of Fig 3.2).

(a) A terminal MSC node is transformed into a sub-LTS collecting the linear event sequence from the scenario. For a MSC M, the LTS captures the set of traces $\mathcal{L}_{total}(M)$ defined in Section 2.4.1.

(b) For a node expanded into a finer-grained g-hMSC, the procedure is applied recursively to obtain the corresponding sub-LTS.

(c) A decision node is rewritten as a sub-LTS reduced to one simple state and no event. The same applies to the *start* and *end* nodes of the g-hMSC.

In cases (a) and (b), the special events $Task_{start}$ and $Task_{end}$ are added as first and last events of the corresponding sub-LTS, respectively.

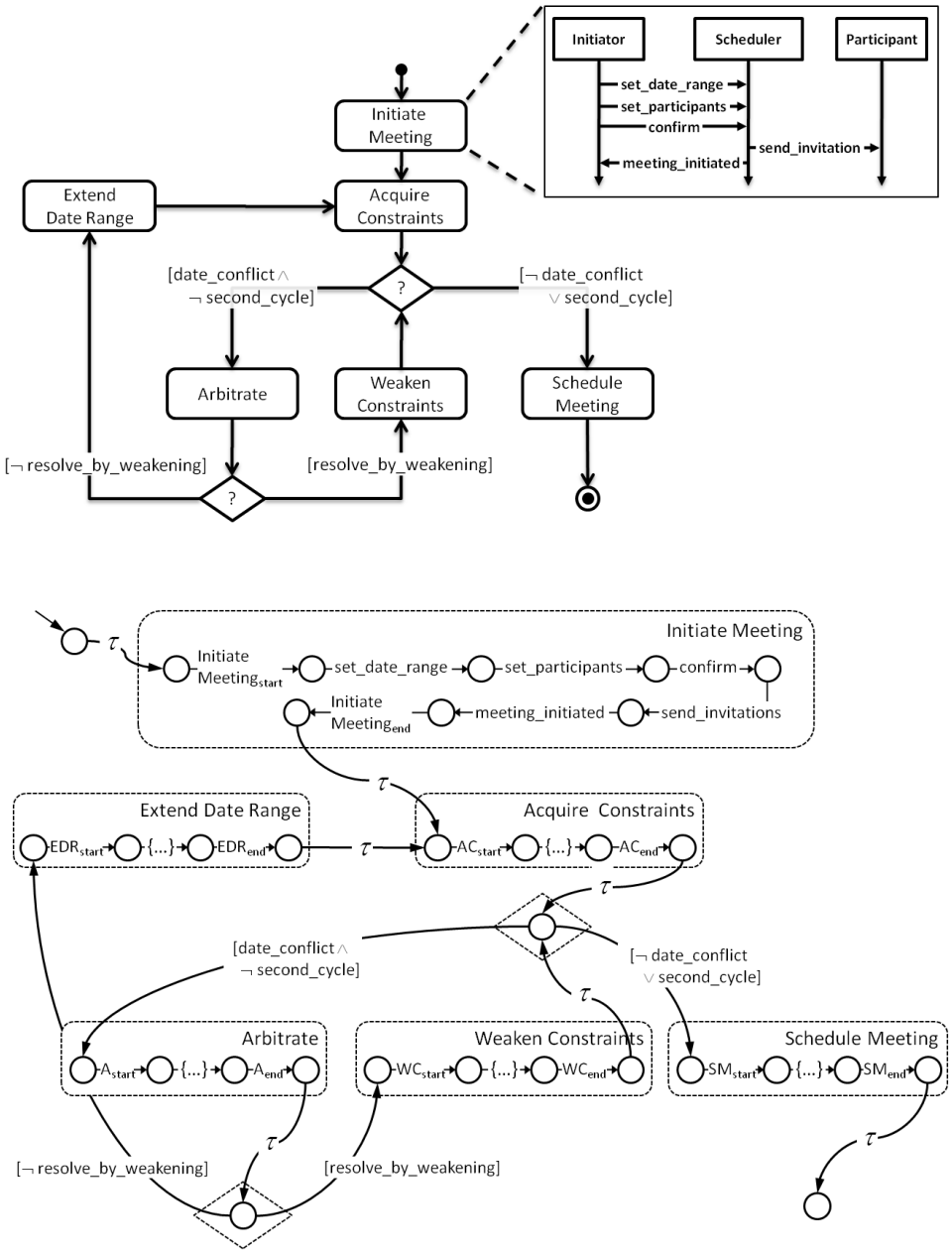


Figure 3.2: Transforming a g-hMSC (top) into a g-LTS (bottom): the meeting scheduler example

Handling edges – The edges in the g-hMSC yield transitions between the *terminal* and *initial* states of the sub-LTS corresponding to their *source* and

target nodes, respectively.

- An outgoing edge of a decision node is further labeled by the corresponding guard to yield a guarded transition in the g-LTS.
- Any other edge is simply converted to a τ transition.

Fig. 3.2 illustrates the transformation on the meeting scheduling process. The g-hMSC on top yields the g-LTS at the bottom.

- In the g-LTS, task "boundaries" are kept visible through dashed rectangles surrounding the different sub-LTS. Similarly, dashed diamonds show which g-LTS states come from decision nodes.
- The task *Initiate Meeting* is converted into a sub-LTS capturing the linear event sequence defined by its MSC. This sequence is extended with the special events $InitiateMeeting_{start}$ and $InitiateMeeting_{end}$.
- Converting the other tasks is similar, the algorithm being applied recursively on finer-grained g-hMSC. The figure shows that *start* and *end* events are introduced for each of those tasks as well.
- Last, outgoing edges of decision nodes lead to guarded transitions in the g-LTS whereas other edges of the g-hMSC yield τ transitions.

The resulting g-LTS can be further optimized by coalescing states separated by τ transitions, provided the target state has no other incoming transition. The g-LTS state space is thereby reduced; this will speed up the synthesis of a trace-equivalent LTS and the model-checking procedure described in Section 7.1.

3.3.2 From guarded LTS to pure LTS

The set of traces accepted by a g-LTS G may now be captured by a trace-equivalent LTS. There are two steps.

(1) First, a parallel composition of g-LTSs is computed so as to meet the various acceptance conditions from Definition 3.3 (valid g-LTS execution).

- The first automaton in this composition is a super g-LTS built from G to initially meet the *trace inclusion* and *admissible start* conditions. This g-LTS actually covers a superset of all admissible event traces and will therefore be pruned.

- To meet the *guard satisfaction* condition, the set of traces of the super g-LTS is pruned further by composing it with fluent automata, one for each fluent.

(2) By construction, all paths admitted by the g-LTS resulting from step (1) yield valid executions for the initial g-LTS G. Therefore, guards and τ-transitions can be safely hidden to obtain a pure LTS; the latter captures all event traces of the trace semantics given in Definition 3.6.

Let us make each g-LTS in the composition further precise. In the sequel, $G = (Q, \Sigma, \Phi, \delta, q_0, C_0)$ will denote the input g-LTS which is transformed to a pure LTS, e.g. the one of Fig. 3.2.

Super g-LTS

By definition, the input g-LTS G already meets the *trace inclusion* condition: the latter simply requires executions to correspond to existing paths in the g-LTS.

In order to meet the *admissible start* condition, G is extended by converting its initial condition C_0 into an explicit guard from a new initial state. More precisely, the super g-LTS is defined by:

$$Super\ LTS = (Q \cup \{q_{start}\}, \Sigma, \Phi, \delta \cup \{(q_{start}, C_0, q_0)\}, q_{start}, true)$$

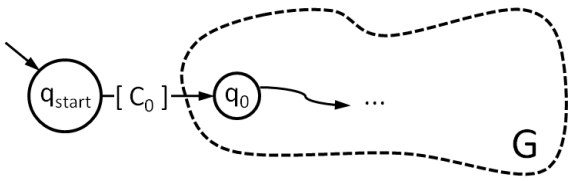


Figure 3.3: The super g-LTS

The super g-LTS is illustrated in Fig 3.3. The cloud at there simply denotes the g-LTS G taken as input. A guarded transition labeled by C_0 reaches its initial state q_0 from a newly introduced one q_{start}.

During composition, the introduced guarded transition will synchronize with similar guarded transitions in the fluent automata introduced hereafter. Doing so will prune initial fluent assignments that do not satisfy C_0. This will guarantee the satisfaction of the *admissible start* condition in a way similar to the satisfaction of guards (see below).

Fluent g-LTS

The *guard satisfaction* condition is enforced by pruning all traces violating guards in the super g-LTS. For this, the super g-LTS is composed with fluent automata that keep track of current fluent values and constrain guards accordingly. Recall that fluents have undertermined initial values; process models and guarded LTS have an initial condition C_0 that constrains their values in the initial state.

Every fluent $Fl = <Init_{Fl}, Term_{Fl}>$ yields a g-LTS $(Q, \Sigma, \Phi, \delta, q_0, C_0)$ where:

$$
\begin{aligned}
Q &= \{q_u, q_t, q_f\} \\
\Sigma &= Init_{Fl} \cup Term_{Fl} \\
\Phi &= \{Fl\} \\
\delta &= \{\ (q_f, e, q_t) \mid e \in Init_{Fl}\ \} \cup \{\ (q_t, e, q_t) \mid e \in Init_{Fl}\ \} \\
&\quad \cup\ \{\ (q_t, e, q_f) \mid e \in Term_{Fl}\ \} \cup \{\ (q_f, e, q_f) \mid e \in Term_{Fl}\ \} \\
&\quad \cup\ \{\ (q_u, [Fl], q_t),\ (q_u, [\neg Fl], q_f)\ \} \\
&\quad \cup\ \{\ (q_t, [Fl], q_t),\ (q_f, [\neg Fl], q_f)\ \} \\
q_0 &= q_u \\
C_0 &= true
\end{aligned}
$$

In this definition, q_u, q_t and q_f denote g-LTS states where the fluent is *unassigned*, *true* and *false*, respectively (see Fig. 3.4).

- The unassigned state q_u is the initial g-LTS state. From there, the *true* and *false* states may be reached through guarded transitions that force the fluent to an initial value consistent with the reached state. These guards will synchronize with the transition labeled by C_0 in the super g-LTS, thereby meeting the *admissible start* condition.

- A fluent g-LTS may transit from its *false* state q_f to its *true* state q_t with the occurrence of events belonging to $Init_{Fl}$ and vice-versa, with events belonging to $Term_{Fl}$. In Fig. 3.4, a transition labeled with a set between brackets is a shortcut denoting one transition for each element of the set.

- In each state q_f and q_t, a self-looping guard constrains the fluent value to the one denoted by the corresponding state. The conjunction of such guards with the ones of the super g-LTS will force the resulting composition to meet the *guard satisfaction* condition. Indeed, the g-LTS composition operator prunes the conjunctions of guards that are unsatisfiable (see Definition 3.8).

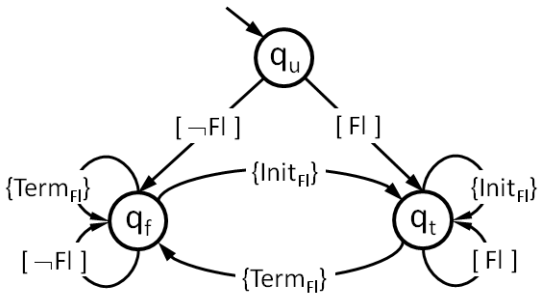


Figure 3.4: A generic fluent g-LTS

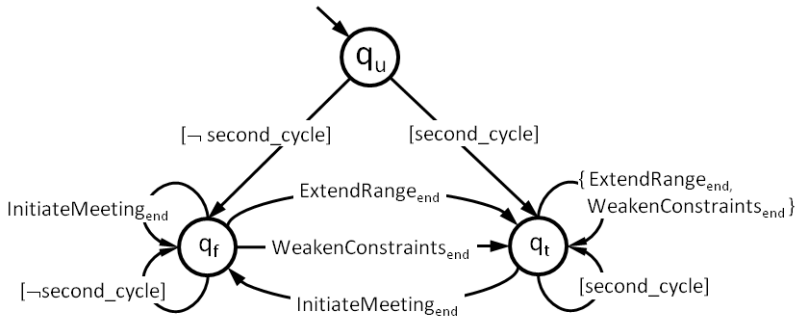


Figure 3.5: Fluent g-LTS for *second_cycle*

- The remaining self-looping transitions labeled with $Init_{Fl}$ (resp. $Term_{Fl}$) from states q_t (resp. q_f) is added to avoid over-constraining the composition. That is, initiating (resp. terminating) events may occur even when the fluent is already *true* (resp. *false*).

For example, consider the following fluent for the meeting scheduler exemplar:

$$\text{fluent } second\_cycle = \\ <\{ExtendDateRange_{end}, WeakenConstraints_{end}\}, \\ \{InitiateMeeting_{end}\}>$$

The corresponding fluent automaton is shown in Fig. 3.5.

Composition automata and hiding guard/τ labels

Putting pieces together, the trace-equivalent LTS of a g-LTS is obtained through the following computation:

$$Composition = (Super\ LTS \parallel Fl_1 \parallel \ldots \parallel Fl_n)$$
$$Traces = Composition \setminus \mathcal{P}(2^{\Phi}),$$

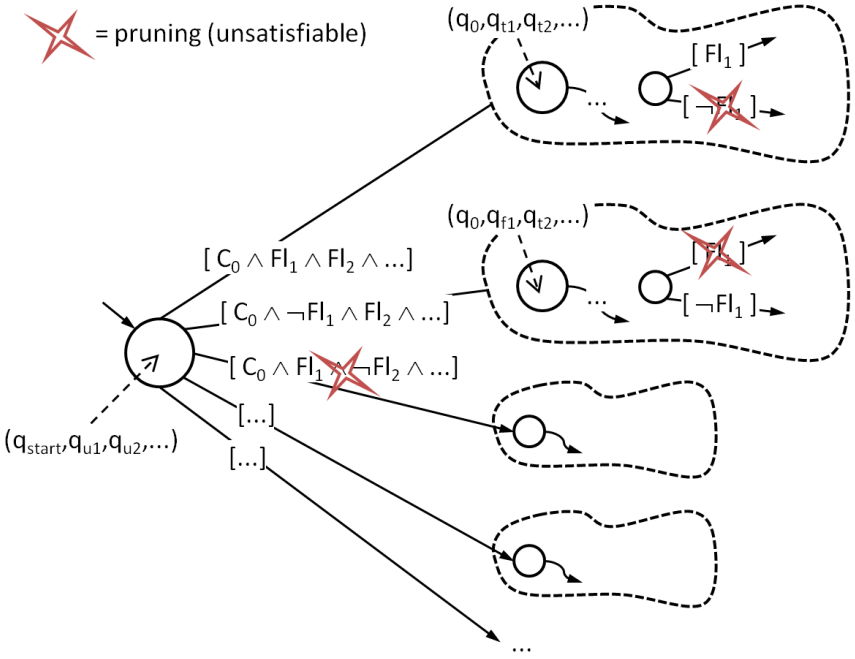


Figure 3.6: Principle of the composition algorithm for computing the valid traces of a g-LTS.

where Fl_i denotes the fluent automaton of the i-th fluent.

The first step simply computes the parallel composition of the Super LTS with fluent automata. All guards are then removed from the results through g-LTS hiding, yielding a pure LTS (see Definition 3.7). This LTS may be further simplified by removing τ transitions and minimizing it under trace equivalence (see Section 2.3).

Correctness of the synthesis

The principle of the synthesis algorithm is illustrated in Fig. 3.6, showing the g-LTS obtained when composing the super g-LTS with fluent automata:

- From the compound initial state, up to $2^{|\Phi|}$ guarded transitions may appear. They actually correspond to all possible fluent assignments satisfying C_0. Unsatisfiable conjunctions are automatically pruned by the g-LTS composition operator (see Definition 3.8).

- Each of these transitions leads to a new compound state, namely q_0 on the super g-LTS side and q_t or q_f for each fluent automaton, according to the initial fluent assignment reaching the compound state.

- From every such state, the input g-LTS G is systematically explored; unsatisfiable guards are pruned according to the current fluent values. The latter are tracked by fluent automata that transit from q_f to q_t with the occurrence of initiating and terminating events.

Admissible paths from the initial state of such automaton yield sequences of transition labels of the following form:

$$<g_0\ l_0\ l_1\ l_2\ l_3\ \ldots\ l_n>$$

where g_0 is a guard and l_i is either a guard or an event label. By construction, g_0 actually reduces to a fluent value assignment, that is, a Boolean guard with only one admissible value assignment. Therefore, such sequences denote g-LTS executions (see Definition 3.2).

We can now show that these executions are valid g-LTS executions for the input g-LTS G (see Definition 3.3):

- The *trace inclusion* condition is met since the sub-sequence $<l_0\ \ldots\ l_n>$ denotes an existing path in G. In particular, fluent automata can only prune guards; they could not add behaviors to those already admissible by the input g-LTS.
- The *admissible start* condition is also met since g_0 has been shown to satisfy C_0.
- The *admissible guards* condition is met since unsatisfiable guards are systematically pruned during composition.

The *hiding* step removes all guards, yielding a pure LTS that precisely captures the trace semantics from Definition 3.6.

In practice, it is convenient to enhance the *hiding* step so as to preserve the initial fluent value assignments g_0 labeling the transitions issued of the initial g-LTS state. This amounts capturing the set of valid g-LTS traces from Definition 3.5. Doing so allows providing the user with a better feedback when using the synthesis algorithm for model-checking g-hMSC and g-LTS (see Section 7.1).

Example

Figure 3.7 shows the resulting LTS obtained on the meeting scheduling example from Fig. 3.2. The task refinements are not entirely unfolded there; only events used in fluent definitions were kept.

- The *no_remaining_solution* event, for example, denotes an initiating event of the *date_conflict* fluent; it belongs to the refinement of the *AcquireConstraints* task.

- The events *click_extend* and *click_weaken* suggest buttons allowing the meeting initiator to select resolution heuristics when a conflict occurs. They appear in the definition of the fluent *resolve_by_weakening*.
- A transition labeled with '...' denotes a sequence of events not shown for readability.

Unlike the original g-hMSC and its g-LTS reformulation, this LTS does not contain any loop. This happens because the *second_cycle* fluent prevents indefinite meeting arbitration. In other words, in our example the technique unfolds the process models into three possible scenarios.

- In the rightmost one, no date conflict arises and the meeting is scheduled immediately after having acquired the constraints.
- In the middle one, a conflict occurs and is resolved by weakening the participant constraints.
- In the leftmost one, the resolution consists in extending the date range and collecting constraints one again.
- In the last two cases, even if the absence of an optimal solution triggers a new date conflict, the meeting is scheduled.

A note on temporal complexity

By construction, our composition technique always considers $2^{|\Phi|}$ initial fluent assignments, pruning those that do not satisfy C_0. In practice, certain fluents have a specific initial value; in this case, the corresponding fluent automaton can be optimized by removing a transition from its initial state. This allows avoiding to compute some of the unsatisfiable conjunctions on the compound initial state.

The approach might even be improved so as to consider only fluent initializations satisfying C_0. If such optimization speeds up the computation in practice, the exponential blow of the resulting trace LTS remains unavoidable. It naturally results from the ability of models with guards to cover numerous behaviors in an implicit, compact way.

Summary

This chapter has shown how state machines can be derived from process models captured in the g-hMSC language. The main objective of this transformation is to make such processes amenable to analysis such as model-checking. For this, a trace-based semantics for g-hMSC has been defined.

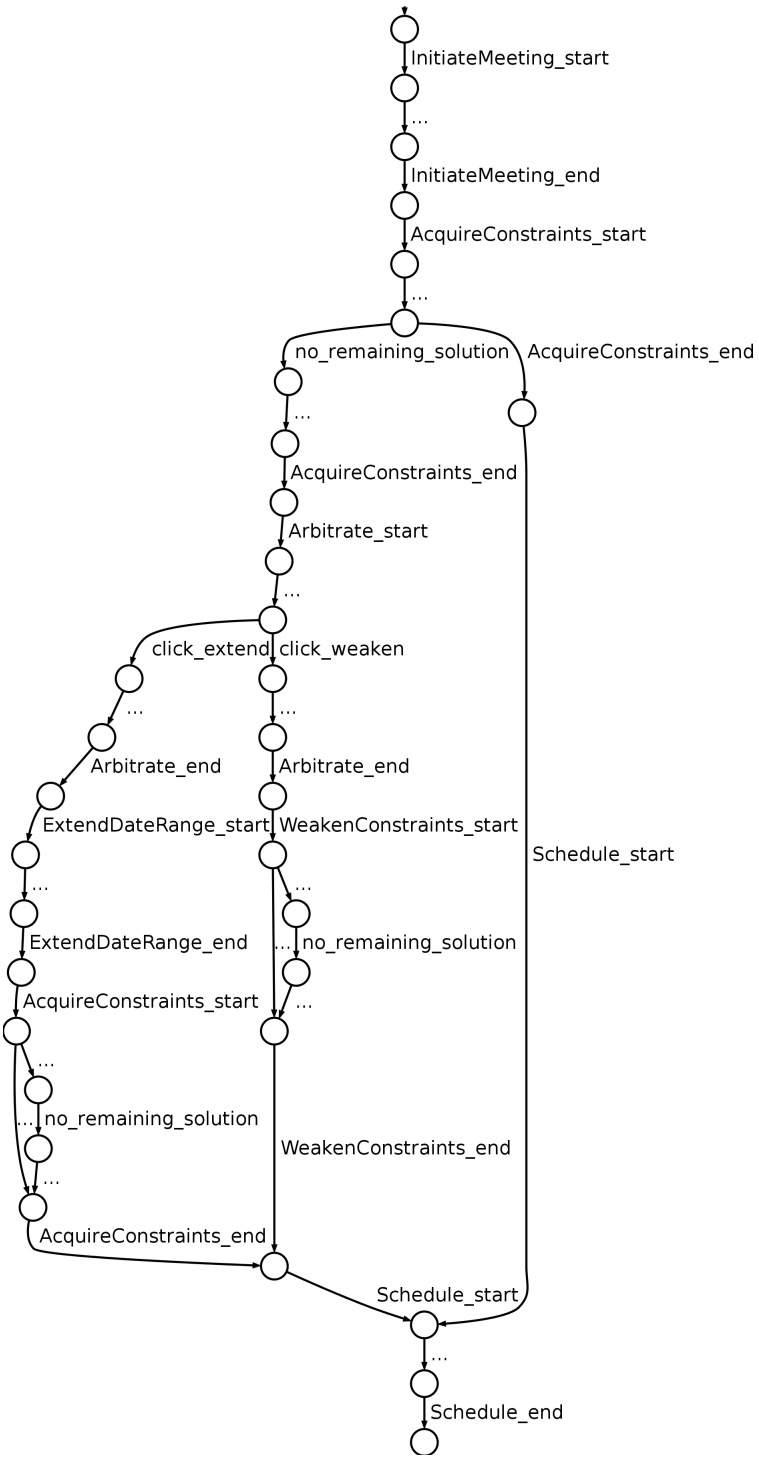


Figure 3.7: Trace-equivalent LTS for the meeting scheduling example from Fig. 2.2.

This semantics is defined in terms of labeled transition systems. A guarded flavor of LTS has been introduced as a powerful intermediate form in the transformation from guarded hMSC to pure LTS; g-LTS are a structured form of LTS avoiding state explosion; they enable a variety of symbolic analyses involving guards [Dam09, Dam11]. Synthesis algorithms from g-hMSC to g-LTS to LTS have been discussed.

Chapter 4

Inductive Synthesis of State Machines from Scenarios

This chapter presents an inductive approach for synthesizing state machines from scenarios. Section 4.1 characterizes the problem, discusses a few requirements and provides an overview of our solution. Section 4.2 provides some required background on grammar induction, the inductive framework on which our techniques are rooted [Gol78]. Section 4.3 describes a technique for learning LTS models from collections of MSCs. This technique is interactive; the end-user is expected to classify additional scenarios generated by the technique as positive or negative examples of system behavior. In Section 4.4, fluent, goals and domain properties are injected in the process to enforce inter-model consistency and prune the inductive search space for better performance. Section 4.5 discusses how hMSCs can be used as richer input of the synthesis process. Section 4.6 discusses the correctness of our approach.

4.1 Objectives and approach

This section provides a characterization of the synthesis problem and an overview of our solution. Section 4.1.1 makes this problem more precise in terms of the models from Chapter 2. Section 4.1.2 states additional requirements on the synthesis approach; the problem statement can be strengthened to take each of them into account. Section 4.1.3 presents an overview of our inductive approach in the light of these requirements.

4.1.1 Problem statement

In its simplest form, the problem of synthesizing LTS state machines from MSC scenarios can be stated as follows:

> Given a consistent MSC collection showing typical examples and counterexamples of system behavior:
>
> $$Sc = (S^+, S^-),$$
>
> synthesize the system as a composition of agent LTSs:
>
> $$System = (Ag_1 \parallel \ldots \parallel Ag_n),$$
>
> such that Sc and $System$ are consistent.

In accordance with our framework hypotheses in Chapter 2, we implicitly assume that the synthesized system is represented through a minimal state machine. The consistency condition covers the three following criteria, discussed in Section 2.4.5.

Structural consistency: the state machine and scenario views must be *structurally* consistent, that is, they must agree on the agent decomposition and their respective interface.

Consistent agent view: each timeline of the positive scenarios in S^+ must specify an existing path in the corresponding agent state machine. Similarly, each timeline of the positive precondition of a negative scenario in S^- must specify an existing path in the corresponding agent state machine.

Consistent system view: the system state machine must cover all positive scenarios and the preconditions of all negatives ones; it must also exclude all negative scenarios. The precise conditions from Section 2.4.5 are:

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(System) \tag{4.1}$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(System) = \emptyset, \tag{4.2}$$

where $\mathcal{L}^+(Sc)$ and $\mathcal{L}^-(Sc)$ encode the scenario collection in terms of positive and negative event traces, respectively. We implicitly assume here the most general case where a partial ordering among MSC events is used.

4.1.2 Requirements on the synthesis approach

The above characterization provides a precise and minimal statement in terms of the formal models of Chapter 2. It may be further completed with the following requirements.

Behavior generalization – Scenarios provide only *examples* of system behaviors; they are thus incomplete. The synthesized state machines should cover more behaviors than those provided as input.

Covering more behaviors is allowed by our formal statement but not enforced. The *consistent system view* condition states an upper bound on behavior generalization by requiring negative scenarios to be excluded. This upper bound needs to be refined if other models are to be taken into account (see below).

Incremental synthesis – Rather than requiring the whole scenario collection to be available at the beginning, the synthesis should accommodate new scenario examples and counterexamples incrementally as they arise during the synthesis process.

- End-users are most likely unable to provide a comprehensive set of scenarios from the beginning, possibly including state assertions along scenario episodes or flowcharts on such episodes. The synthesis approach should therefore work even if only a few scenarios are available when the synthesis starts.
- Both positive and negative scenarios should be taken into account. Negative scenarios appear fairly common in the material provided by stakeholders. They naturally illustrate violations of implicit goals in a way that might be easier to specify.
- In an interactive approach, a more comprehensive set of scenarios is likely to be *eventually* available. The synthesis should smoothly accommodate new scenarios and further information arising in the later phases of the synthesis. For example, the synthesis should be able to take a hMSC as input.

Multi-view synthesis – The possibility of "richer" input specifications should be taken into account in the synthesis process, including for example fluent definitions or goal specifications. Such specifications are not *required* as input; they may however improve the synthesis when available.

Our problem statement must therefore be strengthened:

- All models used as input at some stage or another of the synthesis are required to be consistent with each other.

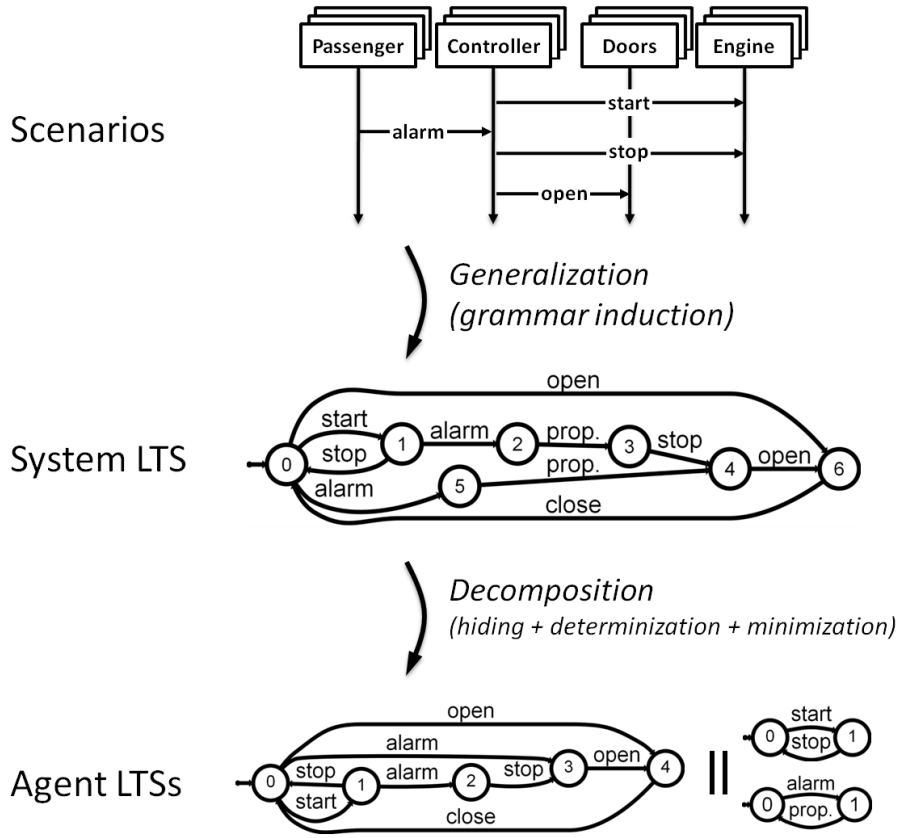


Figure 4.1: Inductive LTS synthesis from MSCs: overview

- The synthesized output system itself must be consistent with all such models. In particular, the synthesized system may not violate goals if the latter are provided as input.

The incremental integration of those requirements in our approach will drive the structure of this chapter. The next section provides an overview of the synthesis process.

4.1.3 Overview of our inductive approach to synthesis

Figure 4.1 shows the two main steps of our approach for the simple case where a scenario collection is taken as input.

The generalization step

This step extends grammar induction techniques developed in [Onc92] to synthesize a system LTS covering all positive traces and excluding all negative ones. The generalization strategy is borrowed from the RPNI algorithm (RPNI stands for Regular Positive and Negative Inference) developed in the context of grammar induction [Onc92], see Section 4.2 hereafter. Roughly, it proceeds as follows.

An initial LTS A_0 is built as a first solution; it has the shape of a tree that accepts all positive scenario traces but does not generalize behaviors. Next, to achieve such generalization, this initial LTS is incrementally refined into automata $A_1, \ldots, A_n$ by merging well-chosen state pairs (see Fig. 4.4 on page 76 for an illustration of this process).

This process is such that the two relations below hold. Relation (4.3) states that the initial LTS A_0 is a valid solution when it comes to accepting all positive traces. Relation (4.4) states that the generalization process is monotonic in that the behaviors accepted by an intermediate solution will remain accepted by every of its successors, up to the final LTS A_n.

$$\mathcal{L}^+(Sc) = \mathcal{L}(A_0) \tag{4.3}$$

$$\mathcal{L}(A_0) \subseteq \mathcal{L}(A_1) \subseteq \ldots \subseteq \mathcal{L}(A_n) \tag{4.4}$$

To avoid overgeneralization, this process is performed under the control of the negative traces from $\mathcal{L}^-(Sc)$. Every intermediate solution A_i has to meet the *consistent system view* condition; the latter provides a useful algorithm invariant, namely,

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(A_i) \tag{4.5}$$

$$\mathcal{L}^-(Sc) \cap \mathcal{L}(A_i) = \emptyset \tag{4.6}$$

This invariant provides a proof argument when compared to relations (4.1) and (4.2). Correctness arguments will be detailed in Section 4.6.

The decomposition step

In this second step, a LTS for each agent is computed by projecting the system LTS onto their respective alphabet. For an agent Ag the projection of the system LTS S is given by:

$$(S \setminus \Sigma^c_{Ag})^{\Delta} \tag{4.7}$$

where Σ^c_{Ag} denotes the set of all system events excluding those of Ag's interface.

The above formula makes use of the hiding, determinization and minimization operators on LTS discussed in Section 2.3.

Meeting additional requirements

As we just mentioned it, the generalization step is driven by RPNI; Section 4.2 provides some background on this grammar induction approach. However, this is just a first milestone toward an inductive LTS synthesis approach that meets all requirements in Section 4.1.2.

- Scenario behaviors are generalized without requiring additional state or flowchart information.
- RPNI does not support incremental system synthesis; multi-model consistency is not guaranteed either.

Section 4.3 will therefore introduce the Query-driven State Merging (QSM) algorithm that extends RPNI with features for user interaction and incremental generalization. QSM supports the elicitation of additional, "interesting" scenarios that were not originally supplied by the end-user. The original collection of scenarios is completed by asking the user scenario queries that are generated during synthesis. A *scenario query* consists of showing the user a specific scenario, asking her to classify it as positive or negative. Answers to such queries guide the generalization process towards more accurate state machines while incrementally enriching the initial scenario collection.

Our multi-view synthesis requirement is tackled in Section 4.4. QSM is extended there so as to allow the injection of additional information that constrains the induction and prunes the inductive search space for better performance. Such additional information may include global definitions of fluents; declarative domain properties; behavior models of external components; and goals that the system is expected to satisfy. The extended QSM thereby guarantees the consistency of synthesized state machines with all other models.

Managing the consistency of a large scenario collection is challenging; hMSCs partly address this through a structured form of scenarios (see Section 2.4.4). Section 4.5 shows how hMSCs can be added to scenario collections as input of the generalization process. In this setting, negative scenarios and other models can still be used to constrain induction so as to achieve multi-view consistency. Accepting hMSCs as input also allows us to remove the requirement stating that all input scenarios start in the same system state.

4.2 Grammar induction for LTS synthesis

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 lan-

guage; 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. For a regular language L, $A(L)$ will denote its canonical automaton, that is the DFA having the smallest number of states that accepts L. We know from [Hop79] that $A(L)$ is unique up to a renumbering of its states.

4.2.1 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 [Gol67]. The strings are randomly taken 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 explains 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 [Gol67]. In practice, convergence in finite time towards an exact solution is often bargained with reasonably fast convergence towards a good approximate solution [Lan92].

4.2.2 The search space of DFA induction

We will use the common notations from formal language theory. Let Σ denote a finite alphabet; a, b, c denote elements of Σ, sometimes called *symbols*; u, v, w denote *strings* over Σ and λ denotes 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.

Symbols here correspond to event labels in Chapter 2 whereas strings correspond to traces. The definition of finite automaton is recalled hereafter; remember that a LTS is a special case of finite automaton where all states are accepting states. Section 4.2.4 will discuss the reduction of LTS synthesis to DFA induction in more detail.

Definition 4.1 (Finite automaton)**.** *A finite automaton is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$ *where* Q *is a finite set of states,* Σ *is an alphabet,* δ *is a transition function mapping* $Q \times \Sigma$ *to* 2^Q*,* q_0 *is the initial state, and* F *is a subset of* Q *identifying the accepting states. The automaton is called a DFA if for any* q *in* Q *and any* e *in* Σ*,* $\delta(q, e)$ *has at most one element.*

Generalizing a positive sample S_+ can be performed by merging states from an initial automaton that only accepts it. This initial automaton is

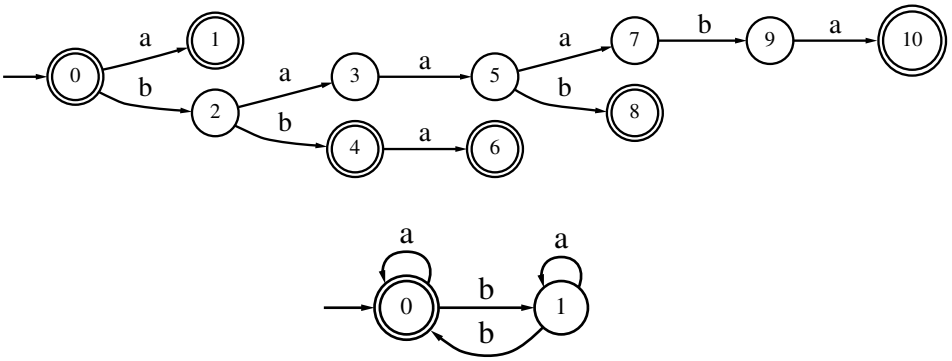


Figure 4.2: $PTA(S_+)$ where $S_+ = \{\lambda, a, bb, bba, baab, baaaba\}$ is a structurally complete sample for the canonical automaton $A(L)$ shown at the bottom. $A(L) = PTA(S_+)/\pi$ with $\pi = \{\{0, 1, 4, 6, 8, 10\}, \{2, 3, 5, 7, 9\}\}$.

called a prefix tree acceptor; it is denoted by $PTA(S_+)$. It is the largest DFA[1] exactly accepting S_+ (see Fig. 4.2). The generalization operation is formally defined through the concept of *quotient automaton*.

Definition 4.2 (Quotient automaton [Dup94])**.** *Given an automaton A and a partition π defined on its state set, the quotient automaton A/π is obtained by merging all states q belonging to the same partition subset $B(q, \pi)$. A state $B(q, \pi)$ in A/π thus corresponds to a subset of states in A.*

A state $B(q, \pi)$ is accepting in A/π if and only if at least one state of $B(q, \pi)$ is accepting in A. Similarly, there is a transition on the symbol a *from state $B(q, \pi)$ to state $B(q', \pi)$ in A/π if and only if there is a transition on* a *from at least one state of $B(q, \pi)$ to at least one state of $B(q', \pi)$ in A.*

By construction of a quotient automaton, any accepting path in A is also an accepting path in A/π. Therefore, for any partition π of the state set of A, $L(A) \subseteq L(A/\pi)$. Merging states in an automaton thus generalizes the accepted language.

A regular language can be learnt if S_+ is representative enough of the unknown language L and if an adequate space of possible generalizations is searched through. These notions are stated precisely hereafter.

Definition 4.3 (Structural completeness [Dup94])**.** *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*

[1] More precisely, the largest *trimmed* DFA, that is, a DFA in which all states are reachable from the initial state.

at least once and every final state is used as accepting state of at least one string.

Rather than a requirement on the sample, structural completeness should be considered as a limit on the possible generalizations 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; this solution should therefore be discarded.

Theorem 4.1 (DFA search space [Dup94])**.** *If a positive sample S_+ is structurally complete with respect to a canonical automaton $A(L)$, there exists a partition of the state set of $PTA(S_+)$ such that $PTA(S_+)/\pi = A(L)$.*

This theorem defines the search space of the DFA induction problem as the set of all automata that can be obtained by merging states of the PTA. Some automata in this space are not deterministic; an efficient determinization process can however enforce the solution to be a DFA (see Section 4.3).

Fig. 4.2 presents the prefix tree acceptor (top) built from the sample $S_+ = \{\lambda, a, bb, bba, baab, baaaba\}$ which is structurally complete with respect to the canonical automaton (bottom). This automaton is a quotient of the PTA for the partition $\pi = \{\{0, 1, 4, 6, 8, 10\}, \{2, 3, 5, 7, 9\}\}$ of its state set.

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

The size[2] of this search space makes any enumeration algorithm unfeasible for any practical purpose. Moreover, finding a minimal consistent DFA is known to be a NP-complete problem [Gol78, Ang78]. Interestingly, the RPNI algorithm or the QSM algorithm described in Section 4.3 only search through a fraction of this space.

4.2.3 Characteristic samples for the RPNI algorithm

The RPNI algorithm is not fully detailed here; the original version is a particular case of our interactive QSM algorithm, see Section 4.3. The convergence of RPNI towards the correct automaton $A(L)$ is guaranteed

[2]Let n be the number of states of $PTA(S_+)$. Let $||S||$ denote the sum of the lengths of the strings in a sample S. By construction, $n \in \mathcal{O}(||S_+||)$. The size of a search space is the number of ways a set of n elements can be partitioned into nonempty subsets. This is called a Bell number $B(n)$. It can be defined through Dobinski's formula: $B(n) = \frac{1}{e}\sum_{k=0}^{\infty}\frac{k^n}{n!}$. This function grows much faster than 2^n.

when the algorithm receives an input sample that includes a *characteristic sample* of the target language [Onc92]. A proof of convergence is presented in [Onc93] in the more general case of transducer learning. Some further notions are needed here.

Definition 4.4 (Short prefixes and suffixes)**.** *Let $Pr(L)$ denote the set of prefixes of L, with $Pr(L) = \{u|\exists v, uv \in L\}$. The right-quotient of L by u, or set of suffixes of u in L, is defined by $L/u = \{v|uv \in L\}$. The set of short prefixes $Sp(L)$ of L is defined by $Sp(L) = \{x \in Pr(L)|\neg\exists u \in \Sigma^* \text{ with } L/u = L/x \text{ and } u < x\}$.*

In a canonical automaton $A(L)$ of language L, the set of short prefixes is the set of first strings in standard order[3], each leading to a particular state of the canonical automaton. As a consequence, there are as many short prefixes as states in $A(L)$. In other words, the short prefixes uniquely identify the states of $A(L)$. The set of short prefixes of the canonical automaton of Fig. 4.2 is $Sp(L) = \{\lambda, b\}$.

Definition 4.5 (Language kernel)**.** *The kernel $N(L)$ of language L is defined as $N(L) = \{xa|x \in Sp(L), a \in \Sigma, xa \in Pr(L)\} \cup \{\lambda\}$.*

The kernel of a language is thus made of its short prefixes extended by one symbol together with the empty string. By construction $Sp(L) \subseteq N(L)$. The kernel elements capture the transitions of the canonical automaton $A(L)$. Indeed, they are obtained by adding one symbol to the short prefixes which capture its states. The kernel of the language defined by the canonical automaton of Fig. 4.2 is $N(L) = \{\lambda, a, b, ba, bb\}$.

Definition 4.6 (Characteristic sample)**.** *A sample $S^c = (S^c_+, S^c_-)$ is characteristic for the language L and the algorithm RPNI if it satisfies the following conditions:*

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

Condition 1 states that each element of the kernel belongs to S^c_+, if it also belongs to the language, or is prefix of a string of S^c_+ otherwise. This condition implies the structural completeness of the sample S^c_+ with respect to $A(L)$. In this case, Theorem 4.1 guarantees that the automaton $A(L)$ can be derived by merging states from $PTA(S^c_+)$.

[3]The standard order of strings on the alphabet $\Sigma = \{a, b\}$ is $\lambda < a < b < aa < ab < ba < bb < aaa < \ldots$

When an element x of the short prefixes and an element y of the kernel do not have the same set of suffixes ($L/x \neq L/y$), they necessarily correspond to distinct states in the canonical automaton. In this case, Condition 2 guarantees that a suffix u would distinguish them. In other words, the merging of a state corresponding to a short prefix x in $PTA(S_+^c)$ with another state corresponding to an element y of the kernel is made incompatible by the existence of xu in S_+^c and yu in S_-^c, or the converse.

To sum up, good examples for learning a canonical automaton $A(L)$ allow the merging of non-equivalent states q and q' to be avoided. Such good examples are the short prefixes of q and q' concatenated with the same suffix u to form a positive example from one state and a negative example from the other.

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

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

For the language accepted by the canonical automaton in Fig. 4.2, one can check that $S = (S_+, S_-)$, with $S_+ = \{\lambda, a, bb, bba, baab, baaaba\}$ and $S_- = \{b, ab, aba\}$, forms a characteristic sample

The definition of a characteristic sample given above may appear quite strong. It is however the standard definition for the RPNI algorithm [Onc92, Dup96]. The definition is based on a worst-case analysis that does not make full use of the exact order in which state pairs are considered during the merging process. It does not rely on a specific order between the symbols of the alphabet. As observed in the experiments described in Chapter 5, a fraction of such sample is often enough to produce approximate state machines with a very high generalization accuracy for randomly generated target DFAs. This observation is also consistent with the results reported in [Lan98].

4.2.4 LTS synthesis as RPNI induction

The synthesis of a system LTS from MSC collections can be reduced to a grammar induction problem as follows.

Consider a scenario collection $Sc = (S^+, S^-)$. $\mathcal{L}^+(Sc)$ denotes the set of positive traces extracted from positive scenarios and from the preconditions of the negative ones; $\mathcal{L}^-(Sc)$ denotes the set of negative traces extracted

from the negative scenarios (see Chapter 2). Both denote finite sets of finite sequences over an alphabet Σ. This holds both under partial and total ordering of MSC events. Also remember that $\mathcal{L}^+(Sc)$ is prefix-closed, that is, the prefixes of positive traces are positive traces as well.

$\mathcal{L}^+(Sc)$ and $\mathcal{L}^-(Sc)$ form valid samples for grammar induction. They can be used as positive and negative input samples of the RPNI algorithm, respectively. As $\mathcal{L}^+(Sc)$ is prefix-closed, the regular language learned by RPNI will be prefix-closed as well. Therefore the resulting DFA is a LTS; it contains only accepting states.

This system LTS covers all positive traces from $\mathcal{L}^+(Sc)$ and excludes all negative ones from $\mathcal{L}^-(Sc)$. In other words, it covers all positive scenarios and rejects all negative scenarios from Sc. Therefore, the system LTS and the scenario collection Sc are consistent.

The next section details this approach on QSM, our interactive variant of the RPNI algorithm.

4.3 Interactive LTS synthesis from MSC collections

This section introduces QSM, a Query driven State-Merging synthesis technique. Its specification is as follows:

Given

- A consistent MSC collection showing typical examples and counterexamples of system behavior:

$$Sc = (S^+, S^-)$$

- An *oracle* correctly classifying system traces as positive or negative examples of desired system behavior,

synthesize a LTS capturing system behaviors,

$$System,$$

such that

- Sc is correctly extended with the answers to scenario questions by the oracle, and
- *System* is consistent with the extended scenario collection.

The specification of QSM is similar to the synthesis problem statement in Section 4.1.1. The main difference comes from the presence of the oracle and the fact that the decomposition step is not implemented by QSM itself. Moreover, remember from Section 2.4.5 that requiring the consistency of the input scenario collection implies that all scenarios start in the same system state.

Algorithm 1 gives the pseudo-code of the QSM algorithm. RPNI can be seen as a particular instance without the inner-most while loop. Roughly, the induction process starts by constructing an initial LTS covering all positive scenarios only. The LTS 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 from the initial LTS. The process is such that, at any step, the current LTS is consistent with all positive scenarios and all negative ones, including the interactively classified ones. This process terminates when no state pairs can still be considered for merging.

In the sequel, two states will be said *compatible* for merging (resp. *incompatible*) if the quotient LTS resulting from their merging is consistent (resp. inconsistent) with the current scenario collection.

Algorithm 1: QSM: interactive LTS synthesis based on an input scenario sample and scenario queries

```
A ← Initialize(Sc)                                              1
while (q, q') ← ChooseStatePair(A) do                           2
    A_new ← Merge(A, q, q')                                     3
    if Consistent(A_new, Sc) then                               4
        while Query ← GenerateQuery(A, A_new) do                5
            if CheckWithEndUser(Query) then                     6
                Sc ← (S+ ∪ {Query}, S−)                         7
            else                                                8
                Sc ← (S+, S− ∪ {Query})                         9
                return QSM(Sc)                                  10
        A ← A_new                                               11
return A, Sc                                                    12
```

In Algorithm 1, the `Initialize` function returns an initial candidate LTS built from the scenario collection. This function computes $\mathcal{L}^+(Sc)$, the set of positive scenario traces, according to the trace semantics defined in Chapter 2. Those traces are captured through a PTA; as the positive sample is prefix-closed, this PTA has all accepting states.

Next, pairs of states are iteratively chosen from the current solution according to the `ChooseStatePair` function[4]. The quotient automaton obtained by merging such states, and possibly some additional states, is computed by the `Merge` function. By construction, the obtained quotient automaton covers all positive scenarios (see Definition 4.2). Its consistency with available negative scenarios is then checked by the `Consistent` function. The various functions are more precisely specified hereafter.

When consistent, new scenarios are generated through the `GenerateQuery` function. These scenarios are submitted to the end-user for classification (see Section 4.3.2). The scenario collection is then refined with these scenarios according to their classification. If all generated scenarios are classified as positive, the quotient automaton becomes the current candidate solution. The process is iterated until no more pair of states can be considered for merging. When a generated scenario is classified as negative, the algorithm is recursively called on the extended scenario collection.

Section 4.3.1 describes the general process of merging compatible state pairs while Section 4.3.2 focuses on the generation of queries submitted to the end-user. Section 4.3.3 discusses an optimization of the search order implemented in `ChooseStatePair`, known as the the Blue-Fringe strategy [Lan98].

4.3.1 Merging compatible state pairs

The various functions that control how merging is performed from an initial automaton are specified hereafter.

Initialize The `Initialize` function returns the PTA built for $\mathcal{L}^+(S)$. Remember that this set includes traces from both the positive scenarios and the preconditions of the negative ones. The PTA built from the initial scenario collection in Figure 4.3 is shown on top of Figure 4.4. For simplicity, these scenarios define a total order on their events; in other words, they admit one linearization only (see Section 2.4).

ChooseStatePair The candidate solution has to be refined by merging well-selected state pairs. The `ChooseStatePair` function determines what pairs to consider. It relies on the standard order on strings. Each state of the PTA can be labeled by its unique prefix from the initial state. Since prefixes can be sorted according to that order, the states can be ranked accordingly. For example, the PTA states in Fig. 4.4 are

[4]The assignment in the corresponding **while** loop is assumed to be *true* whenever a valid state pair is returned by `ChooseStatePairs`. When no more state pairs are considered for merging, `ChooseStatePairs` returns *(nil,nil)* and the assignment is evaluated to *false.* This abuse of notation in the pseudo-code allows to be more concise. A similar remark also applies to the inner **while** loop of the QSM algorithm.

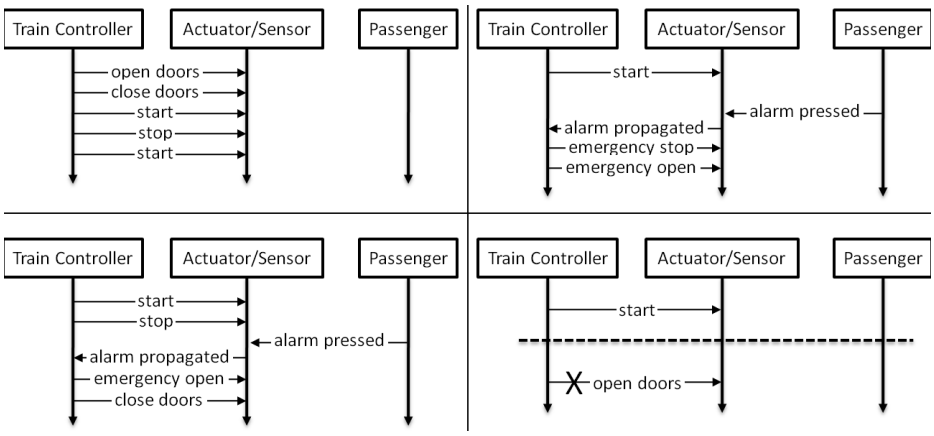


Figure 4.3: Initial positive and negative scenarios for a train system.

labeled by their rank according to this order. The algorithm considers states q of the PTA in increasing order. The state pairs considered for merging only involve such state q and any state q' of lower rank. The q' states are considered in increasing order as well. This particular ordering is specific to the original RPNI algorithm.

Merge The `Merge` function merges the two selected states (q, q') in order to compute a quotient automaton, thereby generalizing the current set of positive behaviors.

In the example of Fig. 4.4, we assume that states 0, 1, and 2 were previously determined not to be compatible for merging. This information typically comes from negative scenarios initially submitted or generated scenarios that were rejected by the user.

Merging a candidate state pair may produce a non-deterministic LTS. For example, after having merged $q = 3$ and $q' = 0$ in the upper part of Fig. 4.4, two transitions labeled `start` from state 0 lead to states 2 and 6, respectively. In such case, the `Merge` function will merge states 2 and 6 and, recursively, any further pair of states that introduces non-determinism.

This recursive operation of removing non-determinism will be called *merging for determinization*. This operation guarantees that the current solution at any step is deterministic. As a result, it produces an automaton which may accept a more general language than the one it starts from. Therefore, it is not equivalent to the standard algorithm for transforming a non-deterministic automaton into a deterministic one accepting the same language [Hop79]. Notably, the time complexity of merging for determinization is a *linear* function of the number of states of the automaton it starts from. In contrast, standard de-

terminization is exponential in the worst case. Furthermore, the resulting automaton is still part of the same inductive search space (see Section 4.2.2).

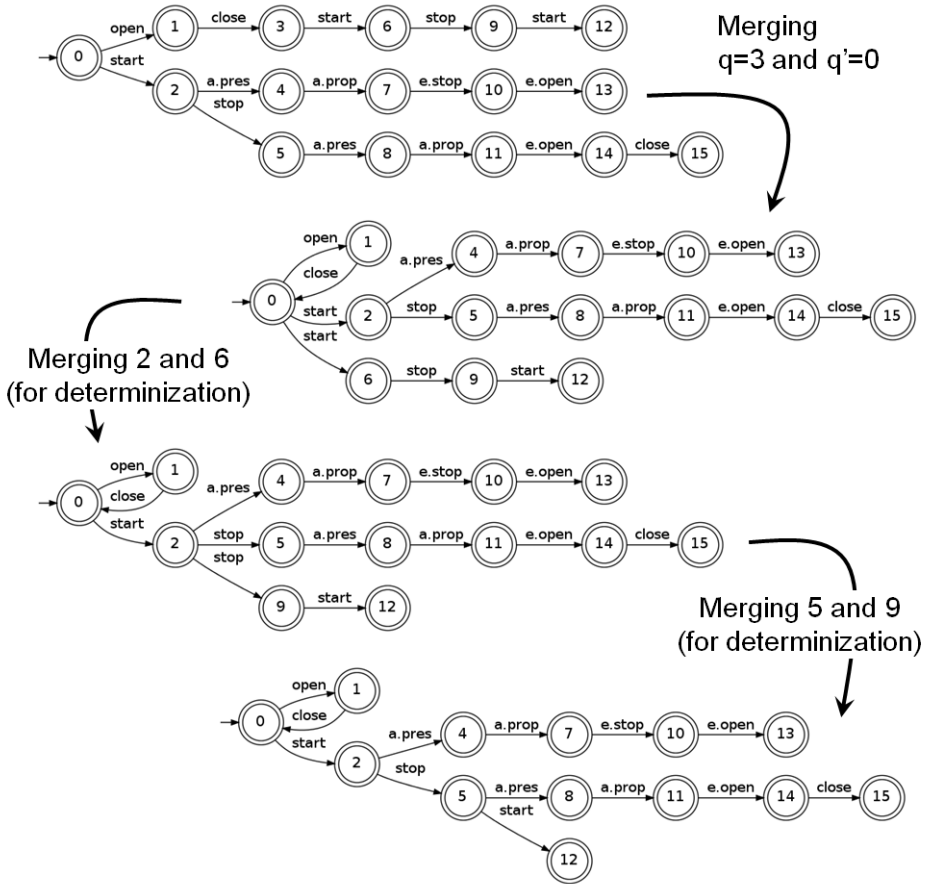


Figure 4.4: Induction step of the QSM algorithm; `a.pres`, `a.prop`, `e.open`, `e.stop` stand for `alarm pressed`, `alarm propagated`, `emergency open`, `emergency stop`, respectively.

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 string ordering, state q is said to be *consolidated*. In the example, states 0, 1, and 2 are consolidated.

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

be consistent.

4.3.2 Generating queries submitted to the end-user

This section describes how queries are generated in the QSM algorithm and how the end-user answers are processed.

GenerateQuery When an intermediate solution is consistent with the available scenarios, new scenarios are generated for classification by the end-user as positive or negative. The aim is to avoid overgeneralization by enriching the possibly limited collection of initial scenarios. The notion of characteristic sample drives the identification of which new scenarios should be generated as queries.

As we saw it in Section 4.2.3, a sample is characteristic of a regular language L if it contains enough positive and negative information. On the one hand, the required positive information is the set of short prefixes $Sp(L)$ which form the shortest histories leading to each state of the canonical automaton $A(L)$. This positive information must also include all elements of the kernel $N(L)$; they represent all system transitions, that is, all shortest histories followed by any admissible event. If such positive information is available, $A(L)$ can always be derived from the PTA by an appropriate set of merging operations. On the other hand, the negative traces provide the necessary information to make incompatible the merging of states that should be kept distinct. A negative trace which would exclude the merging of a state pair (q, q') can simply be made of the shortest history leading to q' followed by any continuation from state q, as detailed below.

Consider the current solution of our induction algorithm when a pair of states (q, q') is selected for merging (line 5 in Algorithm 1). By construction, q' is always a consolidated state at this step of the algorithm; that is, q' is considered to be in $Sp(L)$. State q is always both the root of a tree and the child of a consolidated state. In other words, q is situated at one symbol of a consolidated state – that is, q is considered to be in $N(L)$. States q and q' are compatible according to the available negative scenarios; they would be merged by the standard RPNI algorithm.

The QSM extension will first confirm or infirm the compatibility of q and q' by generating scenarios to be classified by the end-user. The generated scenarios are constructed as follows.

Let A denote the current solution, $L(A)$ the language generated by A, and A_{new} the quotient automaton computed by the `Merge` function at some given step. Let $x \in Sp(L)$ and $y \in N(L)$ denote the short

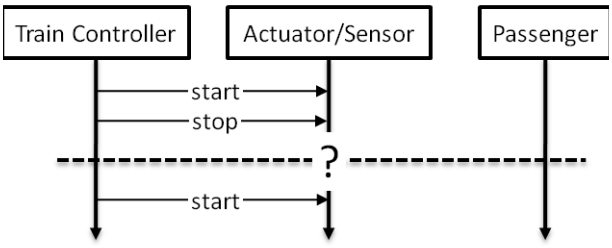


Figure 4.5: A new scenario to be classified by the end-user.

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 built from a system trace xu such that $xu \in L(A_{new}) \setminus L(A)$. It can be further decomposed as xvw such that $xv \in L(A)$. The trace 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 system trace can be converted in a MSC using the structural information provided by a context diagram [Jac95]. The 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. 4.4 with selected state pair ($q = 3, q' = 0$). As q' is the root of the PTA, its short prefix is the empty trace λ. The suffixes of q here yield a simple generated question (see Fig. 4.5), which can be rephrased as follows: "*when having started and stopped the train, can the controller restart it?*". As we can see, the first episode of this scenario in Fig. 4.4 is already accepted by A whereas the entire behavior is accepted in A_{new}.

CheckWithEndUser Whenever a new scenario is generated, it is submitted as a query to the end-user. If the end-user classifies the *Query* as positive, it is added to the collection of positive scenarios. This addition extends the search space as it extends S^+ and consequently the PTA. However, this extension is implicit as the new solution A_{new} is, by construction, also a quotient automaton of this extended PTA. When the *Query* is classified as negative, the induction process is recursively started on the extended scenario collection.

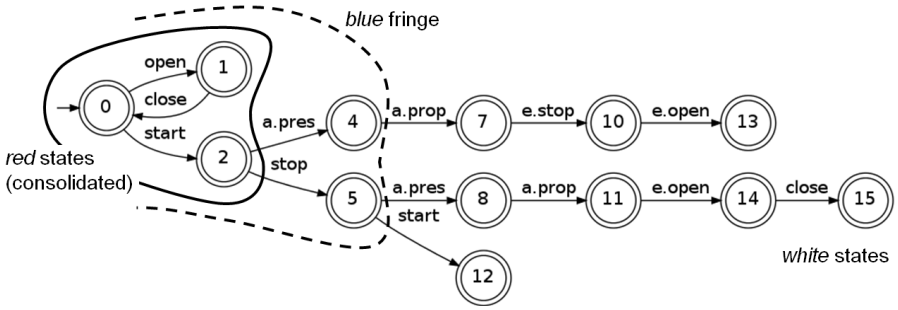


Figure 4.6: Consolidated states (also called "red states") and states on the fringe ("blue states") in a temporary solution.

The QSM algorithm has a polynomial time complexity in the size of the learning sample, see Section 4.3.4. When it receives a characteristic sample in the initial scenario collection, no additional scenario can be classified as negative. As a consequence, QSM will not be called recursively anymore; it stops by returning the target model.

Chapter 5 will detail an experimental study of the actual sample size required to observe the convergence of QSM and the number of queries submitted to the end-user.

4.3.3 Reducing the number of queries: the blue-fringe optimization

The order in which states are considered for merging by `ChooseStatePair` in Section 4.3.1 follows from the implicit assumption that the current sample is characteristic. 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 in case the initial sample is sparse and actually not characteristic for the target system LTS.

To overcome this problem, an optimized strategy known as Blue-Fringe [Lan98] can be used. The difference lies in the way state pairs are considered for merging. The general idea is to detect incompatible state pairs early and, subsequently, to first consider state pairs for which compatibility has the highest chance to be confirmed by the user through positive classification. The resulting "please confirm" interaction may also appear more appealing to the user.

Fig. 4.6 gives a typical example of a temporary solution produced by the original algorithm. Three state classes can be distinguished in this LTS. The so-called "red" states are the consolidated ones (0, 1 and 2 in this example). Outgoing transitions from red states lead to the "blue" states unless the

latter have already been labeled as red. Blue states form the blue fringe (4 and 5 in this case). All other states are called the "white" states.

The original `ChooseStatePair` function in Section 4.3.1 considers the lowest-rank blue state first (state 4 here) for merge with the lowest-rank red state (0). When this choice leads to a compatible quotient automaton, scenarios are generated to the end-user for classification – in this case, a scenario equivalent to the trace `<alarm propagated, emergency stop, emergency open>`.

The above strategy may lead to multiple queries being generated to avoid overgeneralization. Moreover, such queries may be non-intuitive for the user, *e.g.* the `alarm propagated` event is sent to the train controller without having been fired by the `alarm pressed` event to the sensor.

To select a state pair for merging, the Blue-Fringe strategy evaluates all (red, blue) state pairs first. The `ChooseStatePair` function will now call the `Merge` and `Consistent` 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. This is dictated by a scoring mechanism implemented in the `Consistent` function (see below).

When implementing the Blue-Fringe strategy, it appears convenient to adapt `Initialize` so as to build an *augmented* prefix tree acceptor. Such PTA captures the negative traces in $\mathcal{L}^-(Sc)$ in addition to the positive traces in $\mathcal{L}^+(Sc)$. States reached by a negative trace are tagged as error states; they are depicted in black, as in Fig. 4.7.

The `Consistent` function is also updated to return a compatibility score instead of a Boolean value. The score is defined as $-\infty$ when merging the current (red, blue) pair would lead to merging an accepting state and an error state during merging for determinization[5]; this score indicates an incompatible merging. Otherwise, the compatibility score measures how many accepting states have been merged together. The (red, blue) pair with the highest compatibility score is considered first.

The strategy can be further refined with a compatibility threshold α as additional input parameter. Two states are considered to be compatible if their compatibility score is above that threshold. This additional parameter controls the level of generalization since increasing α decreases the number of state pairs that are considered compatible for merging; it therefore decreases the number of generated queries.

[5]in the case of a prefix-closed language, non-error states are all accepting; this is not true for any regular language.

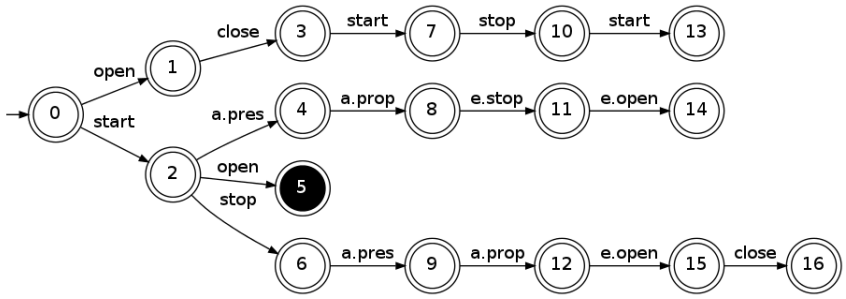


Figure 4.7: Augmented PTA for the scenarios in Fig. 4.3.

Chapter 5 will detail experimental results about the effectiveness of using QSM with and without the Blue-Fringe strategy.

4.3.4 Complexity analysis

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

- Let $n \in \mathcal{O}(||\mathcal{S}_+|| + ||\mathcal{S}_-||)$ denote the number of states of the PTA built from the initial collection of scenarios.
- For a fixed collection of scenarios, there are $\mathcal{O}(n^2)$ state pairs which are considered for merging. The `Merge` and `Consistent` functions have a time complexity linear in n. The `GenerateQuery` can be implemented as a side product of the `Merge` function and does not change its complexity.
- The function `CheckWithEndUser` is assumed to run in constant time.

For a fixed scenario collection, if we abstract from the recursive calls, the time complexity is the same as for the RPNI algorithm; it is upper-bounded by $\mathcal{O}(n^3)$. This bound is obviously not very tight. It assumes that all pairs of states considered by `ChooseStatePairs` appears to be incompatible, which is a very pessimistic assumption. Practical experiments often show that the actual complexity is much closer to the lower bound $\Omega(n)$.

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

Consequently, the number of queries is bounded by $\mathcal{O}(n^2)$ and the global QSM complexity by $\mathcal{O}(n^5)$.

When QSM received a characteristic sample in the initial scenario collection (or any scenario collection considered when calling it recursively), it is guaranteed that no additional scenario can be classified as negative. It follows that QSM will not be called recursively anymore and stops by returning the target model. Note that the size of such a characteristic sample is not necessarily reduced by the fact that any prefix of a positive scenario is also a positive scenario, since the number of negative examples it must contain is not affected by this property.

An experimental study of the actual sample size required to observe the convergence of QSM and the number of queries submitted to the end-user is detailed in Chapter 5.

4.4 Using constraints for multi-view consistency

The interactive QSM algorithm described in Section 4.3 provides a system LTS consistent with all available positive and negative scenarios. The Blue-Fringe strategy can also be applied to reduce the number of additional scenarios submitted to the end-user. The latter strategy relies on two equivalence classes partitioning the states of an augmented PTA. These classes correspond to the accepting states and the error states, respectively. All states belonging to the same class are not necessarily merged in the final solution; however, the `Consistent` function guarantees that only states belonging to the same class *can* be merged.

This approach can be extended to achieve multi-view consistency by incorporating various sources of information. Such information refines the equivalence partition and further constrains the compatible merging operations. Injecting knowledge-based constraints has many advantages:

- It ensures strong consistency of the system LTS with other views;
- It reduces the number of scenario queries in the interactive setting;
- It speeds up the search.

Section 4.4.1 shows how to incorporate domain knowledge such as fluent definitions. Section 4.4.2 shows how goals can be used to constrain the generalization.

The optimization techniques detailed hereafter are based on various equivalence relations over system states. The term *equivalence relation* is used here in its usual mathematical sense, namely, 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.*

4.4.1 Injecting domain knowledge

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;
- Specifications of domain properties.

We discuss them successively.

Propagating fluents

Fluent definitions provide simple and easy-to-provide 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 ($DoorsClosed = false$); it also states 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 decorate each state of the PTA with the value of every fluent. This can be done using a symbolic execution algorithm [Dam06, Dam11] (see Section 2.5).

The pruning rule for constraining the induction process here is to *avoid merging inconsistent states according to these decorations.*

The specific equivalence relation is thus the set of state pairs where both states have *the same fluent value assignment.* The decoration of the merged state is simply inherited from the states being merged.

> *Two states will be considered for merging if they have the same value for every fluent.*

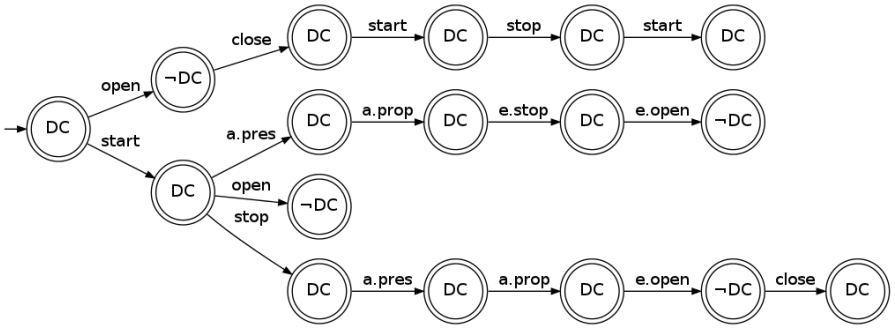


Figure 4.8: Propagating fluents along the PTA to prune the inductive search space (DC stands for *DoorsClosed*)

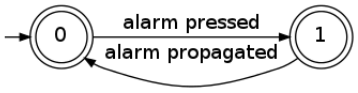


Figure 4.9: LTS model for an alarm sensor.

Fig. 4.8 shows the result of propagating the values of *DoorsClosed* along the augmented PTA built from the scenarios described in Fig. 4.3.

Unfolding models of external components

Quite often the components being modeled need to interact with other components in their environment - *e.g.*, legacy components in a bigger existing system, foreign components in an open system, etc. In such cases the behavior of external components is generally known - typically, through some behavioral model [Hal04]. External components are assumed here to be known by their LTS model.

For example, Fig. 4.9 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 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 onto the PTA. Such decoration is performed by jointly visiting the PTA and the external LTS; the latter synchronizes on shared events and remains in its current state on other events.

Fig. 4.10 shows the result of unfolding the alarm sensor LTS from Fig. 4.9 on the augmented PTA built from the scenarios described in Fig. 4.3. Each state of the PTA is labeled with the number of corresponding states in the alarm-sensor LTS.

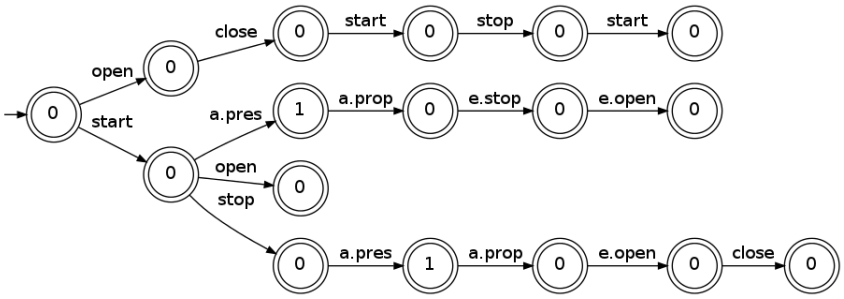


Figure 4.10: Unfolding the alarm sensor LTS onto the PTA; `a.pres` and `a.prop` stand for `alarm pressed` and `alarm propagated`, respectively.

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.*

Using declarative domain properties

Descriptive statements and assumptions about the domain can be expressed declaratively in FLTL (see Section 2.6). For example, the physical law

$$\Box(HighSpeed \rightarrow Moving)$$

excludes all negative traces 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.

4.4.2 Injecting goals

For reasons discussed in Section 2.6, we restrict our attention to goals and domain properties that can be formalized as pure FLTL safety properties. Remember that these properties refer to "*something bad may never happen*".

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

$$\texttt{Maintain[DoorsClosed While Moving]} = \Box(Moving \rightarrow DoorsClosed)$$

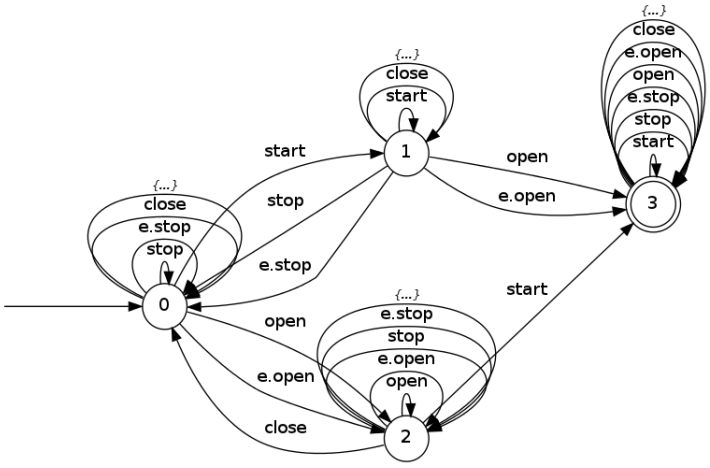


Figure 4.11: Tester LTS for the goal `Maintain[DoorsClosed While Moving]`.

Fig. 4.11 shows the tester automaton for this property (cfr. Section 2.6.3). The accepting state of this tester captures all traces violating the safety property; any trace leading to it corresponds to an undesired system behavior. In particular, the trace `<start, open>` corresponds to the initial negative scenario in Fig. 4.3. As seen in Fig. 4.11, the tester provides many more negative traces. 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. 4.12 shows the PTA decorated using the tester of Fig. 4.11.

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 state of the property tester.

> *Two states will be considered for merging if they have the same property tester state.*

This pruning technique guarantees the consistency between the synthesized system LTS and the considered goals and domain properties. In other words, for every goal or domain property G injected in the synthesis process,

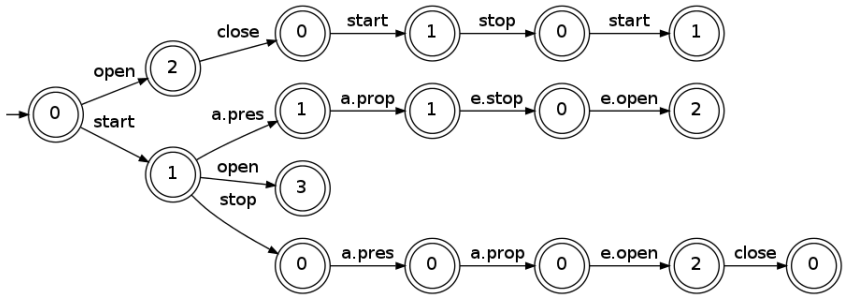


Figure 4.12: Augmented PTA decorated using the tester automaton from Fig. 4.11.

the following consistency condition holds (see Section 2.6.2):

$$\mathcal{L}^{-}(G) \cap \mathcal{L}(System) = \emptyset,$$

where *System* here denotes the synthesized system LTS and $\mathcal{L}^{-}(G)$ captures all traces violating G.

Note that the guarantee given by the above condition is weaker than the *consistent system view* condition (4.2) in Section 4.1.1. The latter requires the consistency of the synthesized system $S = (Ag_1 \parallel \ldots \parallel Ag_n)$ while the above condition only applies to the system LTS. As with implied negative scenarios, a goal could be satisfied by the synthesized System LTS while being violated by the real distributed system. This issue is further discussed in Section 4.7.

4.4.3 Discussion

The equivalence relations considered in the previous sections are all invariant under state merging. In other words, 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.

Our implementation reuses the decoration algorithm from [Dam06] to propagate fluent values on the PTA (see Section 2.5.2). Its generalization in [Dam11] may be used as an effective means for unfolding models of legacy components and tester automata on the PTA without additional developments.

The principle for constraining state merging through equivalence relations first appeared in [Cos98, Cos04]. It can be further instantiated to other equivalence relations not considered here. In particular, it is *not* limited to relations that are invariant under state merging.

As an illustrative example, consider the following generalization of the way fluent values are used to constrain the induction process. We know from Section 2.5.2 that the states of any LTS can be annotated with invariants defined on fluents. Let *inv* denote the function mapping each PTA state to its state invariant; let also denote by *Dom* a domain property that must be met in every state of the system LTS. Consider the following pruning rule:

Two states q and q' will be considered for merging if
$$inv(q) \wedge inv(q') \models Dom$$

The equivalence relation here is the set of state pairs whose conjunction of invariants satisfies the domain property. This equivalence relation is *not* invariant under state merging. The merging constraint can however be enforced; to achieve this, the compound state resulting from merging q and q' has to be annotated by the conjunction of their state invariants; this new invariant is then used in subsequent state merging.

In the general case of DFA induction, in contrast to LTS induction, a similar mechanism is needed to implement the Blue-fringe optimization with an augmented PTA. In that case, an error state may be merged with a non accepting state provided the result is not merged later with an accepting one. That is, the relation capturing the equivalence of states in terms of their continuations is not invariant under state merging.

4.5 LTS synthesis from high-level MSCs

Section 4.1.3 discussed how system behaviors specified in collections of MSC scenarios can be first generalized as a system LTS, then decomposed as a set of agent LTSs. The QSM technique supports the incremental enrichment of an initial scenario collection through scenario queries. It also takes other models into account, such as goals, so as to preserve multi-view consistency. Behavior generalization, incremental synthesis and multi-view consistency were the three main requirements identified in Section 4.1.2.

When it is coupled with other synthesis techniques such as goal mining from scenarios [Dam06], interactive LTS induction appears really effective; starting from a small initial scenario collection, richer system models can be synthesized through a few iterations only. Chapters 5 and 7 will illustrate this claim through evaluations and discussion of tool support.

For non-toy systems, however, large scenario collections might become unmanageable. In particular, the consistency of the collection might be difficult to guarantee without costly refactoring of scenarios. One notable reason is that all scenarios of a collection are required to start in the same system state, as discussed in Section 2.4.3. This may imply a lot of redundancy

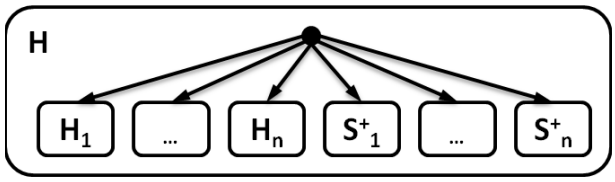


Figure 4.13: Merging multiple hMSCs amounts to building a new hMSC reaching finer-grained hMSCs from its initial state.

in the required input descriptions. Moreover, this constraint may appear arbitrary for end-users who have to draw the scenarios in the first place.

One way to tackle this problem is to use hMSCs for structuring scenario descriptions. As detailed in Section 2.4.4, hMSCs are directed graphs where each node refers to a MSC or a finer grained hMSC (see Fig. 2.10). Scenarios can then be structured by introducing scenario alternatives, sequencing and loops.

A structured form of scenario *helps* specifying a large system with scenarios. It does not *solve* the problem of achieving a complete and consistent view of agent behaviors:

- capturing all possible interleavings of a distributed system proves difficult with scenarios; a single hMSC is therefore hardly complete in practice;
- complementary system features are to be specified in complementary models; in addition to multiple system views, specifying system behaviors in multiple hMSCs makes sense.

A synthesis technique for merging and generalizing behaviors described in hMSCs appears to be a convenient extension to the synthesis technique described in Sections 4.3 and 4.4. To this end, the LTS synthesis statement is first revisited in Section 4.5.1. Our inductive algorithm is then adapted in Section 4.5.2.

4.5.1 Revisiting the LTS synthesis statement

Merging multiple hMSCs $H_1, \ldots, H_n$ with respect to trace behaviors amounts to compute the union of their respective languages. This is equivalent to building a new hMSC H reaching the finer-grained $H_1, \ldots, H_n$ from its initial state. Additional positive scenarios $S_1^+, \ldots, S_n^+$, typically coming from scenario queries, may be integrated in a similar way. This is illustrated in Fig. 4.13.

However, a hMSC only captures positive behavior examples. As explained in previous sections, negative information is needed for avoiding overgeneralization. Negative scenarios, fluents, goals and the like provide a source of negative knowledge that can be used to constrain the induction process. The algorithmic adaptations introduced in the sequel remain compatible with the constraint mechanism based on equivalence relations on system states. This will be detailed in Section 4.5.3.

To keep things simple enough, we may therefore assume that behaviors are specified through one hMSC only, complemented with a scenario collection. The latter contains negative scenarios and answers to scenario queries. Under these assumptions, the LTS synthesis statement is now restated as follows.

> Given a hMSC H and a scenario collection $Sc = (S^+, S^-)$ consistent with each other, that is,
>
> $$[\mathcal{L}^+(Sc) \cup \mathcal{L}(H)] \cap \mathcal{L}^-(Sc) = \emptyset,$$
>
> synthesize the system as a composition of agent LTSs:
>
> $$System = (Ag_1 \parallel \ldots \parallel Ag_n),$$
>
> such that H, Sc and $System$ are consistent with each other.

The selected hMSC trace semantics is not specified in the formulation above. In other words, the set of behaviors captured by $\mathcal{L}(H)$ has to be decided. Remember the relations between the three hMSC languages discussed in Section 2.4.4:

$$\mathcal{L}_{strong}(H) \subseteq \mathcal{L}_{weak}(H) \subseteq \mathcal{L}_{arch}(H) \tag{4.8}$$

$\mathcal{L}_{strong}(H)$ denotes the set of system behaviors with strong sequential composition of hMSC nodes and total event ordering inside MSCs. It is the simplest model. However, it assumes an implicit synchronization scheme to be used by the agents. The latter is usually not available in real distributed systems. $\mathcal{L}_{arch}(H)$ is the most realistic one for such systems as it captures all possible interleavings of agent behaviors. $\mathcal{L}_{weak}(H)$ is mainly used for explaining and detecting implied scenarios in hMSC specifications [Uch03]; it is also the hardest to compute.

Making a choice of semantics is required for generalizing behaviors as inductive synthesis needs a set of traces as input. From an algorithmic point of view, however, the three hMSC languages require the same adaptations of the inductive process, as explained in the next sections.

4.5.2 Generalizing hMSC behaviors

Remember that learning a regular language L aims at generalizing a positive sample S_+ under the control of a negative sample S_- such that the following relation of language inclusions holds:

$$S_+ \quad \subseteq \quad L \quad \subseteq \quad \Sigma^* \setminus S_- \tag{4.9}$$

LTS synthesis from a scenario collection Sc reduces to grammar induction as the sets $\mathcal{L}^+(Sc)$ and $\mathcal{L}^-(Sc)$ are valid positive and negative samples, respectively (see Section 4.2.4). In particular, they denote *finite* sets of traces.

When considering the generalization of hMSC behaviors, the sets of positive and negative traces are $\mathcal{L}^+(Sc) \cup \mathcal{L}(H)$ and $\mathcal{L}^-(Sc)$, respectively. The positive set is no longer a sample as $\mathcal{L}(H)$ might contain an infinite number of traces. Therefore, the current problem statement no longer exactly fits the identification-in-the-limit framework introduced in Section 4.2.

From a theoretical point of view, this means that generalizing hMSC behaviors is a different problem than generalizing MSC behaviors; therefore, further study would be needed to re-state the convergence criterion and the notion of characteristic sample in particular.

From an algorithmic point of view, however, only a few adaptations of RPNI and QSM appear necessary. They are detailed in the next section.

4.5.3 The Automaton State Merging algorithm

The algorithm for generalizing hMSC behaviors is our Algorithm 2, called Automaton State Merging (ASM). As sketched below, its overall structure is very similar to QSM in Section 4.3; the interactive feature is omitted here as it raises a few issues that we discuss later. QSM itself being an interactive extension to RPNI, Algorithm 2 is very similar to RPNI itself which might appear surprising at first glance.

The main difference between RPNI/QSM on one side and ASM on the other side lies in the *initial automaton* built by `Initialize`. RPNI and QSM initially convert the input *sample* as a PTA, hence a tree, whereas ASM converts the input *language* of the hMSC as a LTS, hence a graph.

More precisely, the adaptations required to the different functions of the algorithm are the following:

Initialize This function is adapted to return a LTS instead of a PTA. This LTS is built from two sources:

Algorithm 2: ASM, a state-merging algorithm from high-level Message Sequence Charts

Input: A high-level MSC H and a scenario collection $Sc = (S_+, S_-)$
Output: A System LTS, consistent with both H and Sc

```
A ← Initialize(H, Sc)                                  1
while (q, q') ← ChooseStatePair(A) do                  2
    A_new ← Merge(A, q, q')                            3
    if Consistent(A_new, S_-) then                     4
        A ← A_new                                      5
return A                                               6
```

- On one side, the positive traces from the input hMSC H are captured through a LTS, assuming a suitable choice of hMSC semantics. The choice of $\mathcal{L}_{arch}(H)$ appears natural for distributed systems; in that case, the synthesis algorithm from [Uch03], introduced in Section 2.4.4, is used to synthesize the LTS from the hMSC.
- On the other side, the scenario collection is captured through a PTA, as in RPNI and QSM.

The standard algorithm for capturing the union of regular languages [Hop79] is then used to merge the LTS and the PTA as a single LTS; the latter is returned by *Initialize*.

ChooseStatePair The main generalization loop in ASM does not fold up a PTA anymore, but successively merges the states of an automaton, which might be any graph[6]. In our current ASM implementation, the *ChoosenStatePair* function is slightly adapted to preserve the RPNI search order. ASM pre-computes the natural order among states of the initial LTS solution returned by *Initialize*. A breadth-first search is used; each state is numbered when visited.

Merge From a specification standpoint, the *Merge* function does not require specific adaptations. This results from the mathematical definition of a quotient automaton (see Definition 4.2). The function takes an automaton and two states to merge as arguments; it returns the quotient automaton resulting from this state merging.

From an implementation standpoint, the merging-for-determinization process is often implemented under the assumption of a tree invariant property [Lan98]. This property states that, when considering two

[6] Hence the "Automaton State Merging" name.

states to be merged, at least one of them is the root of a tree. Such property holds for RPNI and QSM, even when the Blue-Fringe optimization is used. It is a sufficient condition for the determinization process to be finite.

Even though it is convenient, the tree invariant property is not required [Lam08]. The main merging loop and the *Merge* function can be implemented without the tree invariant property; indeed, the recursive determinization process stops naturally when non-determinism is completely reduced. In practice, an implementation of RPNI/QSM may require changing some data structures and associated algorithms to be converted to ASM.

Consistent This function does not require specific adaptations either; it is the same as in RPNI or QSM (see Section 4.3.1).

Our current ASM implementation does not go beyond this: it is not optimized with the Blue-fringe heuristics; it does not have the interactive feature of QSM; it is not constrained with domain knowledge. Improvements along these three directions are discussed below.

Blue-fringe heuristics

Adding the Blue-Fringe heuristic to ASM requires two main adaptations:

- The LTS returned by *Initialize* should be augmented with error states encoding the negative strings captured by negative scenarios in the collection. This requires a slight adaptation of the algorithm for computing the union of regular language; this adaptation is needed to handle the error states of an augmented PTA built from the scenario collection.
- In a way similar to the *Merge* function, the distinction made by Blue-fringe between *red* and *blue* states may be implemented with *ChooseStatePair* under the assumption that the initial solution is a PTA (see Section 4.3.3). In that case, another implementation approach might be taken. From a specification standpoint, the characterization of a intermediate automaton in terms of *red*, *blue* and *white* states does not depend on a tree invariant.

Scenario questions

The interactive feature of QSM may be adapted and plugged into ASM. It consists in replacing the main ASM loop by the one of QSM (see Algorithm 1 in Section 4.3). The `GenerateQuery` function is adapted as follows:

GenerateQuery The generation of scenario queries relies on the tree invariant property mentioned above. When merging a state pair (q, q'), a scenario query is built with the shortest trace leading to q concatenated with the suffixes of q'. A QSM invariant is that q' is the root of a tree; this invariant is used to generate finite suffixes for scenario questions (see Section 4.3.2).

If the tree invariant property no longer holds, the `GenerateQuery` function must be extended with a procedure for extracting finite suffixes from q'. This does not introduce any technical problem. For example, pre-computing a spanning tree on the initial LTS associates finite suffixes with each state. What forms a "good" suffix for convergence and scenario classification by end-users remains however an open question. As the adapted algorithm no longer fits the identification-in-the-limit framework, the notion of a characteristic sample would need to be adapted here.

Constraints based on domain knowledge

Even if not supported by our current implementation, the constraint mechanism discussed in Section 4.4 is fully compatible with ASM. Remember that it relies on the partitioning of PTA states according to equivalence relations extracted from domain knowledge.

This partitioning can be performed in a similar way on the LTS returned by `Initialize` in ASM. Unlike a PTA, however, such LTS may already contain cycles. The preconditions of ASM must therefore be strengthened with additional preconditions requiring the consistency between input scenarios and available domain knowledge, such as fluents and models of legacy components. Fluent definitions, for example, must yield a deterministic value for each fluent in every state of the initial LTS returned by *Initialize*.

From an algorithmic standpoint, an effective solution for computing the equivalence relations on the initial LTS relies on specific instantiations of the generic LTS decoration algorithm discussed in [Dam11] (see Section 4.4.3).

4.6 Correctness

This section provides correctness arguments and proofs for the different settings of our inductive approach to state machine synthesis. The simplest approach with RPNI is first discussed; features are then gradually integrated, such as scenario questions and the injection of domain knowledge. A deeper analysis of the problem statement and an overview of this section is first given in Section 4.6.1.

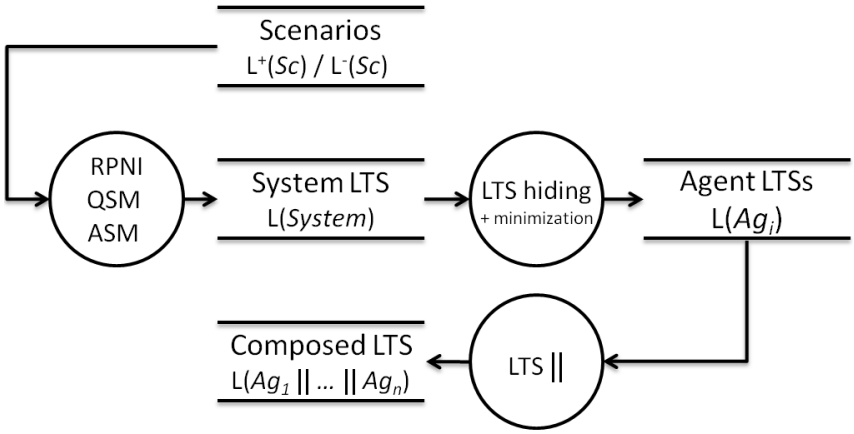


Figure 4.14: Inductive synthesis steps and products.

4.6.1 Overview

In view of the initial problem statement in Section 4.1.1 and its successive strengthening in subsequent sections, proving the correctness of our approach amounts to show that the composed system resulting from synthesis is consistent with input scenarios and all injected domain knowledge.

Fig. 4.14 summarizes our synthesis approach. It shows the algorithms used together with their input/output in terms of behavior models and associated languages. The figure can be read as follows:

- From scenarios, a system LTS is inferred using RPNI, QSM or ASM.
- LTS hiding and minimization is then used to obtain a canonical state machine for each agent.
- From the system point of view, the result of our synthesis approach is precisely captured by the composition of those state machines.

In this figure and in the following discussions,

- $Sc = (S^+, S^-)$ denotes the input scenario collection; its positive and negative languages are denoted by $\mathcal{L}^+(Sc)$ and $\mathcal{L}^-(Sc)$, respectively.

  When considering ASM, Sc denotes both the input hMSC and the scenario collection (see Section 4.5). The positive and negative languages are updated accordingly.
- $\mathcal{L}(System)$ denotes the language captured by the inferred system LTS.
- $\mathcal{L}(Ag)$ denotes the language of an arbitrary agent Ag, as captured by its LTS state machine.

- $\mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n)$ denotes the language captured by the LTS resulting of the composition of the individual agent LTSs.

Remember from Section 4.1.1 that, under the precondition that input models are consistent, three conditions are required to hold on the synthesized state machines:

- The *structural consistency* condition requires input scenarios and synthesized agent state machines to agree on the respective agent interfaces.

  In the case of QSM, structural consistency must also hold for answers to scenario questions. In the case of ASM, this must be the case for the input hMSC as well.

- The *consistent agent view* condition requires each synthesized agent state machine to cover the positive behaviors along the corresponding scenario timeline:

$$\mathcal{L}^+(Sc_{\downarrow Ag}) \subseteq \mathcal{L}(Ag) \text{ for each agent } Ag, \tag{4.10}$$

  where $Sc_{\downarrow Ag}$ denotes the positive behaviors along Ag's timeline in a scenario collection $Sc = (S^+, S^-)$:

$$\mathcal{L}^+(Sc_{\downarrow Ag}) = \bigcup_{P \in S^+} \mathcal{L}(P_{\downarrow Ag}) \;\cup\; \bigcup_{N \in S^-} \mathcal{L}^+(N_{\downarrow Ag}) \tag{4.11}$$

  This is a a slight generalization of the concept of agent traces along the timeline of a single scenario, see $M_{\downarrow Ag}$ in Section 2.4. Observe that $\mathcal{L}^+(Sc_{\downarrow Ag})$ takes both positive and negative scenarios into account.

  In the case of QSM, (4.10) must hold even when Sc is assumed to contain the answers to scenario questions. In the case of ASM, (4.10) remains the same, but the projection of the input hMSC H has to be considered in addition to the scenario collection in (4.11):

$$\mathcal{L}^+(Sc_{\downarrow Ag}) = \bigcup_{P \in S^+} \mathcal{L}(P_{\downarrow Ag}) \;\cup\; \bigcup_{N \in S^-} \mathcal{L}^+(N_{\downarrow Ag}) \;\cup\; \mathcal{L}(H_{\downarrow Ag}) \tag{4.12}$$

- The *consistent system view* requires the composed system to cover all positive scenarios and reject all negative ones:

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) \tag{4.13}$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) = \emptyset \tag{4.14}$$

  In the case of QSM, Sc is assumed to contain answers to scenario questions. In the case of ASM, Sc denotes the input hMSC together with the scenario collection.

When goals are injected in the induction process, this postcondition must be strengthened with the following condition requiring all traces violating the goals to be rejected by the synthesized system:

$$\mathcal{L}^-(Goals) \cap \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) = \emptyset \tag{4.15}$$

where $\mathcal{L}^-(Goals)$ denotes the union of all system traces violating at least one known goal:

$$\mathcal{L}^-(Goals) = \mathcal{L}^-(G_1) \cup \ldots \cup \mathcal{L}^-(G_n) \tag{4.16}$$

The following sections are organized as follows. Section 4.6.2 establishes a first milestone by proving the consistency of the inferred system LTS with scenarios in the case of RPNI. This result is then used in Section 4.6.3 to show that the decomposition step meets the "structural consistency" and the "consistent agent view" conditions. The "consistent system view" condition is further discussed in Section 4.6.4. The discussion is pursued in the presence of scenario questions in Section 4.6.5 and with the injection of goals in Section 4.6.6. Section 4.6.7 closes this section by discussing the correctness of ASM.

4.6.2 Consistency of the system LTS

This section demonstrates that RPNI yields a system LTS consistent with the input scenario collection. The pseudo code given in Algo. 1 is used; the inner-most while loop related to scenario questions is ignored here (lines 5 to 10).

Theorem 4.2. *The system LTS synthesized by RPNI covers all positive scenarios and rejects all negative ones:*

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(System)$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(System) = \emptyset$$

Proof. This theorem is proven by induction using the following invariant:

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(A_i) \tag{4.17}$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(A_i) = \emptyset \tag{4.18}$$

Base: A_0 denotes the PTA returned by *Initialize*.

The PTA is the largest DFA accepting exactly the positive language; an algorithm precondition also states that input scenarios are consistent. Therefore the following conditions hold, entailing the invariant:

$$\mathcal{L}^+(Sc) = \mathcal{L}(A_0)$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(A_0) = \emptyset$$

Inductive step: A_i is the quotient automaton denoting the current solution at step i; (4.17) and (4.18) are assumed to hold on the candidate automaton A_i.

From a solution A_i, the next solution A_{i+1} is computed by the *Merge* function (see line 3 in Algo. 1). The latter computes a quotient automaton of A_i. We prove the positive and negative parts of the invariant in turn.

On one hand, Definition 4.2 guarantees that such quotient automaton may only generalize the language of A_i. As the condition (4.17) is assumed to hold for A_i, the following condition holds as well:

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(A_i) \subseteq \mathcal{L}(A_{i+1})$$

On the other hand, a candidate solution is only kept for the next iteration if it is consistent with the negative scenarios (see lines 4 and 11). Therefore, the following condition holds when such solution is kept (line 11):

$$\mathcal{L}^-(Sc) \cap \mathcal{L}(A_{i+1}) = \emptyset$$

When the algorithm terminates, it returns the last compatible quotient automaton considered.

□

A detailed proof of the convergence of RPNI towards the canonical target automaton when it receives a characteristic sample can be found in [Onc93] in the more general case of transducer learning.

4.6.3 Structural consistency and consistent agent view

Given the consistency of the system LTS with the positive and negative scenarios, the decomposition step guarantees that the *structural consistency* and the *consistent agent view* both hold. We only sketch the structure of such proofs here.

Structural consistency only requires the LTS hiding step to make use of adequate agent alphabets, as induced from the scenarios themselves or given by a (consistent) structural model. We do not discuss it further.

For each agent Ag, the "consistent agent view" condition (4.10) is derived from (4.13) using the following derivations:

$$\mathcal{L}(X) \subseteq \mathcal{L}(Y) \implies \mathcal{L}(X \setminus I) \subseteq \mathcal{L}(Y \setminus I) \tag{4.19}$$

$$\mathcal{L}^+(Sc_{\downarrow Ag}) = \mathcal{L}^+(Sc \setminus \Sigma^c_{Ag}) \tag{4.20}$$

$$\mathcal{L}(Ag) = \mathcal{L}(System \setminus \Sigma^c_{Ag}) \tag{4.21}$$

where Σ^c_{Ag} denotes the set of all system events excluding those of Ag's interface.

- (4.19) states that behavior inclusion is preserved under LTS hiding; this property follows from material in Section 2.3.3.
- (4.20) rewrites the left term of (4.10) in terms of the hiding of scenario behaviors[7]. For RPNI and QSM, it can be derived from (4.11) and the definition of $M_{\downarrow Ag}$ (see Section 2.4.1). For ASM, it can be derived in a similar way from (4.12).
- (4.21) follows from the definition of the decomposition step itself (see (4.7) in Section 4.1.3).

Theorem 4.2 establishing the consistency of the system LTS with input scenarios, the following condition is established thanks to (4.19)

$$\mathcal{L}^+(Sc \setminus \Sigma^c_{Ag}) \subseteq \mathcal{L}(System \setminus \Sigma^c_{Ag}) \tag{4.22}$$

The "consistent agent view" condition (4.10) is established by substituting the right terms of (4.20) and (4.21) in (4.22).

4.6.4 Consistent system view: the problem of implied scenarios

This section discusses the correctness of the "consistent system view" condition in the case of RPNI. The conditions related to the positive and negative scenarios are discussed in turn.

Theorem 4.3. *When using the RPNI induction algorithm, the composed system synthesized by the inductive approach covers all positive scenarios.*

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n)$$

Proof. This condition results from two main properties:

- Theorem 4.2 establishing the consistency of the inferred system LTS:

$$\mathcal{L}^+(Sc) \subseteq \mathcal{L}(System)$$

[7] using an abuse of notation here as the hiding operator is defined on LTS, not on scenarios collections.

- Projecting the system LTS on agent alphabets and composing their LTS afterwards does not restrict behaviors:

$$\mathcal{L}(\text{System}) \subseteq \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n)$$

This latter condition can be derived from the definition of agent languages (4.21) and properties of LTS hiding and composition operators (see Section 2.3).

□

Theorem 4.4 (Candidate)**.** *When using the RPNI induction algorithm, the composed system synthesized by the inductive approach rejects all negative scenarios.*

$$\mathcal{L}^{-}(Sc) \cap \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) = \emptyset$$

Due to the potential presence of implied scenarios, our approach only guarantees the weaker postcondition already given by Theorem 4.2, namely,

$$\mathcal{L}^{-}(Sc) \cap \mathcal{L}(\text{System}) = \emptyset$$

In other words, the induction algorithm ensures that the system LTS excludes all negative scenarios. This property can however be lost after the decomposition and recomposition steps leading to the composed system.

The reason has to be found in the possible occurrence of so-called *implied scenarios* [Alu00, Uch04]. Remember from Section 2.4.4 that implied scenarios may appear when a system is specified system-wide while implemented component-wise.

In our case, the set of implied scenarios is precisely defined as:

$$\mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) \setminus \mathcal{L}(\text{System}) \tag{4.23}$$

that is, implied scenarios denote system behaviors that the system LTS does not accept but which are exhibited by the composition of agent LTSs. Implied scenarios appear when, once distributed, the agents lack monitoring and controlling abilities to restrict their behavior so as to precisely match the system LTS.

Three cases may arise here:

- The set of implied scenarios (4.23) is empty.

  In this case, the inferred system LTS and the composed system coincide. Therefore, Theorem 4.4 is a trivial consequence of Theorem 4.2.

- The set of implied scenarios (4.23) only contains examples of desired system behavior.

  Theorem 4.4 is trivially proven as implied scenarios do not intersect with negative scenarios.

  In this case, the presence of implied scenarios is not problematic and may be seen as the result of a second generalization step due to the decomposition and recomposition of agent LTS. This second generalization step proves useful as it weakens the necessity of having a pure structurally complete scenario collection in the first place.

- The set of implied scenarios (4.23) contains at least one counterexample of desired system behaviors.

  This case is more problematic. In particular, a behavior trace t could be such that the following conditions hold:

  $$t \in \mathcal{L}^{-}(Sc) \tag{4.24}$$
  $$t \notin \mathcal{L}(\text{System}) \tag{4.25}$$
  $$t \in \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) \tag{4.26}$$

  that is, (4.24) t denotes a system behavior explicitly rejected through a negative scenario; it might be a question answered negatively; (4.25) t is correctly rejected by the system LTS; (4.26) t is still be exhibited by the composed system.

  In such case, the candidate Theorem 4.4 is shown too strong. The "consistent system view" condition is not met as (4.14) does not hold. As with other scenario approaches, e.g. [Alu00, Uch04], our synthesis technique fails to guarantee a consistent system view between scenarios and state machines in presence of negative implied scenarios.

  In order to detect such situations, the technique from [Uch04] could be adapted to enumerate implied scenarios and submit them as additional scenario questions to the user (see Section 8.2.3).

  Note that, fixing implied scenario problems requires rethinking the system decomposition into agents and/or refactoring their interfaces. In other words, the root cause of implied scenarios problems has to be found in the structural decomposition of the system, not in the particular technique used to infer state machines from scenarios.

4.6.5 Correctness in the presence of scenario questions

The postcondition of our approach was strengthened in the presence of scenario questions. This strengthening required the synthesized system to be

consistent with all scenario questions in addition to the initial scenario collection.

Provided a consistent system LTS is inferred with QSM, the correctness arguments for the *structural consistency* and *consistent agent view* conditions remain unchanged. The discussion about implied scenarios also takes place here. Therefore, we only prove the consistency of the system LTS induced by QSM when scenario questions are taken into account.

Theorem 4.5. *The system LTS inferred by QSM is consistent with the scenario collection extended with the answers to all scenario questions.*

Proof. This theorem is proven by induction (cfr. Algo. 1):

Base: The base case captures a QSM run where all scenario questions are answered positively.

In such case, QSM roughly reduces to RPNI, for which Theorem 4.2 is known to hold. We still need to prove that all scenario questions are accepted by the synthesized system LTS.

Observe that all scenarios accepted at line 7 are consistent with the current solution A_{new} (line 4). The system LTS returned by QSM is a quotient automaton of A_{new}; Definition 4.2 therefore ensures that the system LTS accepts those scenarios as well.

Inductive step: The inductive step captures a run where a rejected scenario yields a tail recursive call (line 10).

The discussion about positively accepted scenarios remains unchanged. The scenario collection is correctly extended (see line 7).

Every time a scenario question is rejected by the oracle, the scenario collection is correctly extended as well (see line 9). Provided the oracle does not make classification errors, as required in preconditions, the scenario collection remains consistent for the tail recursive call taking place at line 10.

□

4.6.6 Consistency with goals and domain properties

The pre- and postconditions of our approach were strengthened in the presence of goals and domain properties. In precondition, scenarios and goals are required to be consistent. In postcondition, the synthesized system may not violate goals. In other words, provided the input scenarios and safety properties are consistent, the following postcondition is required to hold:

$$\mathcal{L}^{-}(\mathit{Goals}) \cap \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) = \emptyset \tag{4.27}$$

A weaker condition is proven in Theorem 4.6 as implied scenarios may lead to goals being violated after the decomposition and recomposition steps. A few simplifications are in order here:

- We consider here the case of RPNI; in the absence of classification errors by the oracle, the injection of goals is independent of the scenario questions.
- In accordance with the assumptions of the thesis we consider only safety properties (see 2.6).
- Without lost of generality, we consider only one of such safety properties, G.

Lemma 4.1 provides a useful milestone.

Lemma 4.1. *When two states q and q' are considered for merging by QSM, all their prefixes are traces leading to the same state in the tester automaton capturing $\mathcal{L}^-(G)$.*

Proof. We assume the correctness of the joint traversal for annotating the PTA states with their corresponding states in the tester automaton (see Section 4.4.2). The induction algorithm will only consider the merging of state pairs corresponding to the same state in the tester automaton. The prefixes property thus holds for the first merge considered on the PTA; it is trivially preserved under state merging and therefore holds for every state pair considered from the successive quotient automata. □

Theorem 4.6. *Provided G denotes a safety property, the system LTS synthesized by the constrained inductive algorithm is consistent with G.*

$$\mathcal{L}^-(G) \cap \mathcal{L}(\text{System}) = \emptyset$$

Proof. The proof proceeds by induction on the following loop invariant:

$$\mathcal{L}^-(G) \cap \mathcal{L}(A_i) = \emptyset$$

Base: A_0 denotes the PTA.

The invariant holds for A_0 as (1) the preconditions require the scenarios and the goals to be consistent and (2) the PTA does not generalize the positive scenario language.

$$\mathcal{L}^-(G) \cap \mathcal{L}^+(Sc) = \emptyset$$
$$\mathcal{L}(A_0) = \mathcal{L}^+(Sc)$$

Inductive step: Let A_i denote a current solution considered by the algorithm. Suppose the invariant holds for A_i. We show that the invariant holds for A_{i+1}, the quotient automaton of A_i obtained by merging a candidate state pair (q, q').

By construction of the constraint mechanism based on equivalent state classes, q and q' correspond to the same state t in the tester automaton capturing $\mathcal{L}^-(G)$ (see Section 4.4.2).

The tester is known to be a canonical automaton, that is, it is minimal and deterministic (see Section 2.6.3). A bijection thus exists between states and accepted trace suffixes (residual languages) [Hop79].

Therefore, all accepted traces from q (resp. q') in the current solution A_i are rejected traces from t in the tester automaton. By Lemma 4.1 all prefixes of q in A_i (resp. q') are prefixes of t in the tester. As the invariant holds for A_i, their respective suffixes must be disjoint.

When q and q' are merged, the same lemma guarantees that the suffixes "gained" by q' (resp. q) do not yield new traces in A_{i+1} that would violate the safety property G. In other words, the invariant holds for A_{i+1}.

When the algorithm terminates, it returns the last compatible quotient automaton considered.

□

4.6.7 Correctness in the presence of control information

The correctness proof for ASM follows the same reasoning as the one presented for RPNI in Theorem 4.2. Discussions about structural consistent, consistent agent view and implied scenarios remain unchanged.

Theorem 4.7. *The system LTS synthesized by ASM is consistent with the hMSC and the scenario collection taken as input.*

$$[\mathcal{L}(H) \cup \mathcal{L}^+(Sc)] \subseteq \mathcal{L}(System)$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(System) = \emptyset$$

Proof. The theorem is proven by induction, using the following invariant:

$$[\mathcal{L}(H) \cup \mathcal{L}^+(Sc)] \subseteq \mathcal{L}(A_i)$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(A_i) = \emptyset$$

Base: A_0 denotes the automaton returned by *Initialize*.

On one hand, the *Initialize* function implements the synthesis algorithm detailed in [Uch03] which does not generalize hMSC behaviors. On the other hand, the input hMSC and the positive scenarios are required to be consistent with the negative scenarios. Therefore the following condition holds, entailing the invariant:

$$[\mathcal{L}(H) \cup \mathcal{L}^+(Sc)] = \mathcal{L}(A_0)$$
$$\mathcal{L}^-(Sc) \cap \mathcal{L}(A_0) = \emptyset$$

Inductive step: A_i denotes the current candidate solution. The invariant is assumed to hold for A_i.

The main induction loop is similar to the one of RPNI. It considers successive quotient automata, which generalize the language captured by the current solution A_i. Therefore, the following condition holds:

$$[\mathcal{L}(H) \cup \mathcal{L}^+(Sc)] \subseteq \mathcal{L}(A_i) \subseteq \mathcal{L}(A_{i+1})$$

As shown in Algo 2, quotient automata are not kept unless being consistent with the negative scenarios (lines 3 and 4). Therefore, the following condition holds in every case:

$$\mathcal{L}^-(Sc) \cap \mathcal{L}(A_{i+1}) = \emptyset$$

□

4.7 Discussion

Let us step back a little before closing this chapter. This section is aimed at discussing two questions:

- *What problem are we trying to solve?*
- *Why do we solve it in this way?*

In addition to revisiting our synthesis technique and discussing some perspectives, answers to those questions will allow us to compare our approach with alternative ones. Such comparisons will be made in Chapter 8.

Let us start with the former question, "what problem are we trying to solve?".

This chapter discussed an horizontal synthesis technique. Remember from Section 1.3 that horizontal synthesis is aimed at providing automated support for elaborating requirements and exploring system designs. It provides such support through the semi-automated building of missing model

fragments in a multi-view framework; the completion of already available views may be an objective as well.

Along this line, our state machine synthesis technique can be seen as a building block in a broader vision of multi-view behavior modeling. This problem can be stated as follows:

- There is an expected target system, composed of agents $Ag_1, \ldots, Ag_n$ whose behavior can be modeled through state machines. The behavior of the system itself is defined through the composition of the agent behaviors.
- A complete specification of the system behaviors is not available (yet). Behavioral model fragments may however be available; those fragments take various forms:
  - Scenarios may describe examples and counterexamples of desired system behavior.
  - The behavior of some agents may be partially or completely specified through the definitions of agent state variables and/or state machines.
  - Some behavioral goals may specify restrictions on agent and system behaviors so as to avoid undesired system histories.
- The problem to solve is to specify system behaviors more completely. This roughly means completing all models in an incremental, consistent and (hopefully) convergent way.

As just stated, solving the problem above presupposes an incremental approach. Within iterations, system descriptions get more complete and behavior models get richer. Additional system features are added and modeled when successful iterations complete; refactoring steps are conducted when problems appear, such as undesired implied scenarios.

Our inductive state machine synthesis technique is thus only a building block in an integrated approach along this line. Observe that the synthesis of state machines from scenarios can be seen at least partially as a pretext for conducting a broader exploration of the desired system. In the opinionated vision of multi-view modeling above, the synthesis of state machine is seen as equally important as the completion of the scenario collection with answers to scenario questions; the same applies to the identification of behavior goals triggered when some of those scenarios are rejected by the end-user.

The following sections revisit and discuss design choices of our synthesis approach in the light of the multi-view modeling vision above.

4.7.1 Agent behaviors or system behaviors?

The choice of tackling the synthesis of *agent* state machines through the inference of a *system* state machine has not been much motivated so far. Why not inferring one state machine by agent and composing them afterwards, instead of going the other way round?

First, let us stress that such alternative approach makes perfect sense. Our experience in exploring it even suggests that it might provide very good results in terms of behavior generalization. That being said, a few points would need to be adapted and/or taken into account to further explore it.

- Scenario questions would not be available for guiding the inductive process in a per agent approach. The interactive feature might be adapted but it would require the user to be able to classify agent traces. This seems less convenient as it amounts to interact with the end-user in a different language for each agent. Moreover, interactions would not occur in the language used by the end-user for specifying scenarios in the first place.

- Our definition of negative scenarios is borrowed from [Uch02]. Such definition amounts to consider negative system traces only. An effective use of the negative scenarios for the independent synthesis of agent state machines would need this definition to be adapted. Such negative knowledge is required for guiding the induction process towards good generalizations.

- Similarly, our approach considers system-wide goals and domain properties. Unless capturing agent requirements specifically, available safety properties cannot be used to constrain the induction process on a per agent basis. Even in the absence of implied scenarios, the consistency of inferred state machines with goals might be harder to guarantee.

4.7.2 The rationale behind grammar induction

From the specific standpoint of state machine synthesis, our use of grammar induction was motivated by the need to generalize system behaviors described in the scenarios. This choice is further discussed here in the light of the incremental elaboration of requirements discussed in Section 4.7.

To keep things simple enough, the use of fluent state variables and models of legacy components, the structure of scenarios, the decomposition step of our approach, and implied scenarios are ignored aspects here. In other words, we focus here on pure behavioral aspects in the triangle composed of scenarios, state machines and goals [Dam06, Uch07].

- On one side, positive scenarios capture a lower bound on system behaviors. Scenarios capture behaviors that the system *must* exhibit. The enrichment of the scenario language in the transition from QSM to ASM amounts capturing these positive behaviors through a (prefix-closed) regular language. Let $\mathcal{L}^+(Scenarios)$ denote this language.

- On the other side, negative scenarios and goals capture an upper bound on system behaviors. They capture behaviors that the system *may not* exhibit. The system traces violating safety properties are known to be captured by the class of regular languages (see Section 2.6.3). Negative scenarios generally illustrate violations of safety properties; they may be ignored without loss of generality. Let $\mathcal{L}^-(Goals)$ then denote undesired system behaviors.

- Somewhere between those bounds, agent state machines may be seen as capturing the exact behaviors exhibited by the system. Provided those state machines are known, $\mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n)$ denotes those system behaviors.

The consistency of scenarios, state machines and goals are known to be captured by the following conditions (see Chapter 2):

$$\mathcal{L}^+(Scenarios) \subseteq \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n)$$
$$\mathcal{L}^-(Goals) \cap \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) = \emptyset$$

As stated in Section 1.3, an horizontal synthesis technique uses such consistency rules backwards. Instead of checking them, horizontal synthesis uses these rules to semi-automatically synthesize missing models using the information already available in the other views. Our inductive state machine synthesis technique is illustrative of such process. It can be seen as synthesizing missing state machines from scenarios, under the control of goals:

$$\mathcal{L}^+(Scenarios) \;\subseteq\; \mathcal{L}(?) \;\subseteq\; \Sigma^* \setminus \mathcal{L}^-(Goals)$$

The choice of grammar induction methods to tackle the synthesis problem is not incidental. Let us assume that the expected target system is fixed, even if unknown. The problem statement requires the synthesized system to cover all desired behaviors described in the scenarios while rejecting all negative ones captured by the goals. Observe that the LTS capturing exactly the lower bound $\mathcal{L}^+(Scenarios)$ provides a trivial solution to this problem. Using this LTS for specifying system behaviors would however lead to a specification changing every time a new scenario is elicited. In contrast, a grammar induction approach guarantees that beyond a given point, the

inferred specification will not be affected by the addition of new examples and counterexamples of desired system behavior.

In other words, more than offering a practical way of generalizing system behaviors illustrated in the scenarios, our approach provides a guarantee for the convergence towards the target system when available scenarios and goals capture a sufficiently rich knowledge about the target system. Note that such convergence criterion looks necessary even if the synthesis of state machines might be seen as a pretext for elaborating requirements or completing a multi-view system model. We further discuss this claim in the subsequent sections and when comparing our approach with alternative ones in Chapter 8.

It is known that inductive techniques come at a price of an inductive bias [Mit80]. In our case, such bias has to be found in the assumption of structural completeness (see Definition 4.3). Every transition in the target system is expected to be used at least once in the scenarios. Moreover, under this assumption, using grammar induction amounts to search for the *smallest* machine consistent with the available scenarios and goals. In the extreme case where no negative knowledge is available, this state machine is the universal automaton; it has only one state and a self looping transition for every event in the system.

Rather than a requirement on the sample, the assumption of structural completeness must be seen as a limit on the possible generalizations that are allowed from the scenarios. If a transition of the target system is never used in the examples of desired behaviors, no evidence supports its existence and a candidate solution presenting that transition should be discarded. Similarly, rather than a limitation on our approach searching for the smallest consistent machine must be seen as an application of the Occam's principle stating that "among all models explaining the world equally well, the simplest should be preferred".

4.7.3 Perspectives

The discussion in Section 4.7.2 about the convergence of our approach made use of two implicit assumptions.

- The identification-in-the-limit framework provides convergence guarantees only if both positive examples $\mathcal{L}^+(Scenarios)$ and negative examples $\mathcal{L}^-(Goals)$ capture finite sets of traces. The departure from this theoretical framework was already noted when introducing ASM in Section 4.5.2.

  From an grammar induction standpoint, the discussion in Section 4.7.2 goes on step further as it amounts to consider the generalization of a

positive regular language $\mathcal{L}^+$ under the control of a negative regular language $\mathcal{L}^-$.

$$\mathcal{L}^+ \subseteq ? \subseteq \Sigma^* \setminus \mathcal{L}^-$$

An induction algorithm for tackling such problem was proposed in [Lam08], called ASM*. Strictly speaking, the convergence guarantee for the synthesis of state machines from scenarios and goals in Section 4.7.2 would require finding a sound notion of characteristic sample for ASM and ASM*. We are however confident that such a candidate exists.

- Discussing the convergence of an incremental modeling approach only makes sense under the assumption that the expected target system is fixed. Such assumption is hardly realistic in practice. Subsequent modeling cycles introduce additional system features; refactoring steps are often needed as well. Even in such cases, relying on synthesis techniques offering convergence guarantees seems necessary at worse and reasonable at best.

  In any case, techniques such as inductive behavior model synthesis are aimed at being integrated in incremental, multi-view modeling approaches. This supposes that various synthesis techniques are made available in multi-view modeling environments. This raises questions about what constitutes an effective guidance in such environments, and whether such guidance may be supported by additional automated techniques.

  The ISIS tool introduced in Section 7.2 provides such environment around QSM. The visual inspection of synthesized state machines there leads to the definition of negative scenarios and goals when overgeneralization is detected. This typically triggers new modeling cycles and new QSM runs. Using the system knowledge captured by $\mathcal{L}^-(\mathit{Goals})$, the upper bound on system behaviors, opens new perspectives for additional guidance there. A promising approach will be further discussed in Section 8.3.2.

Summary

This chapter discussed how grammar induction may be adapted to synthesize LTS state machines from end-user scenarios. The RPNI algorithm provides a basis to inductively generalize scenario behaviors as a system LTS; the latter is then projected on the alphabet of each agent to obtain their state machines.

QSM extends RPNI with an interactive feature where an end-user classifies generated scenarios as positive or negative examples of desired system behavior. This constrains the induction process towards good behavior generalizations. It also allows completing the initial scenario collection with interesting agent interactions that were not initially explored.

QSM and ASM may be constrained through equivalence relations defined on system states. This mechanism was instantiated to prune the induction process with the definition of fluent state variables, models of legacy components, domain properties, and goals. In addition to guaranteeing the consistency of synthesized state machines with other available models, the injection of such knowledge offers better induction performance and reduces the number of user interactions.

Structured forms of scenario descriptions, such as hMSCs, prove useful for large systems. They overcome a common limitation of using scenario collections, namely, the assumption that all scenarios start in the same system state. The induction of agent state machines from structured forms of scenarios led to the ASM algorithm, another extension of RPNI. While our current ASM implementation is rather limited, the chapter showed that the design of an induction algorithm mixing hMSC input, scenario questions, and injection of domain knowledge and goals raises minor issues only.

The transition from RPNI/QSM to ASM raises interesting perspectives for future research. From a grammar induction standpoint, a further extension called ASM* amounts to consider the generalization of a positive language under the control of a negative one. ASM and ASM* do not exactly fit in the identification-in-the-limit framework; in particular, the convergence criterion would need to be revisited. From a software engineering standpoint, such work would set a sounder basis for tackling the synthesis of behavior models from structured forms of scenarios and safety properties.

Chapter 5

Evaluation

This chapter reports on the results obtained when evaluating our inductive LTS synthesis technique from Chapter 4. Section 5.1 discusses our evaluation objectives and overviews the selected approach. Section 5.2 presents the results of evaluating our algorithms on case studies. These evaluations are complemented by experiments conducted on synthetic datasets whose setup and results are presented in Section 5.3. Our main conclusions are drawn in Section 5.4.

5.1 Objectives and approach

The aim of this chapter is to evaluate our inductive synthesis technique in the light of the thesis objectives. The idea is therefore to check whether our synthesis approach provides an effective way of exploring requirements and conducting system design. Or, in terms of the requirements discussed in Chapter 1,

> *How well does it help building* adequate*,* complete*,* consistent*, and* precise *models for the target system considered?*

Such a question is difficult to answer in absolute terms. Answers can however be provided in two ways:

a) By comparing the technique with existing ones, either theoretically, empirically, or through benchmarks.

b) By using the technique in dedicated experiments. Controlled parameters then provide variation points to conduct comparisons.

This chapter focusses on the second way of conducting such an evaluation. A discussion of how our inductive synthesis approach compares with and/or integrates with existing techniques can be found in Chapter 8.

When our inductive synthesis technique is considered in isolation, the question of how well it helps building high-quality models can already be partially answered. For instance, our technique synthesizes *consistent* models by construction. By design, it also helps *completing* them through scenario questions which in turn trigger the identification of domain properties and goals. Not all questions can be answered so simply:

- How adequate are the synthesized state machines?
- What is the impact of fluent, goal and domain knowledge injection on model adequacy?
- Is the approach usable by end-users?
- How many iterations are needed to obtain models considered complete?
- Does the inductive technique scale and remains usable on large models?

Controlled experiments have thus been conducted to provide answers to those questions. In practice, two kinds of evaluation have been considered, as reflected by the following sections (The specific evaluation protocols used will be described in each case).

- Section 5.2 discusses evaluations conducted on three case studies involving multiple models. The aim here is to evaluate the feasibility of inductive LTS synthesis in practice.

  Our ISIS tool presented in Section 7.2 has been used as an effective support for designing and conducting the evaluations described there. A typical modeling session in ISIS will be illustrated on one of those case studies; additional black-box results will also be detailed.

- Section 5.3 complements this case-driven evaluation with experiments conducted on random automata and samples. The aim here is to study the performance of QSM and ASM in a more systematic way using synthetic datasets whose size grows significantly beyond the average one of the case studies. To achieve sound comparisons, our evaluation protocol inspires from a benchmark known as Abbadingo [Lan98].

  The evaluations conducted here allow us to describe what can be expected from our techniques on systems significantly larger than the case studies considered in Section 5.2. This will also allow us to compare QSM and ASM with state-of-the-art induction algorithms.

The ISIS modeling session described in Section 5.2 provides a general idea about the effectiveness of our multi-view synthesis approach. In addition, controlled parameters of the various experiments provide comparison points to answer finer-grained evaluation questions. Those controlled parameters are:

- The size the target system LTS, from a case-study or one controlled by a random generation procedure.
- The heuristics used for state merging: the RPNI search order or the Blue-fringe optimization.
- The presence of absence of an oracle answering scenario questions.
- The number of fluents and goals injected to prune the induction process.
- The use of control information in scenarios and the richness of such knowledge.

When conducting the experiments, three measures have been collected so as to evaluate results. Those measures have a clear impact on the adequacy and usability of the synthesis technique. Therefore, they allow making the bridge between the controlled parameters and answers to our evaluation questions.

Model adequacy Roughly speaking, *model adequacy* captures *how well* an inferred model matches the expected target behavior model.

Model adequacy is easy to measure in controlled experiments in which, by design, the target model is known. Depending on the experiment, we will use either a binary value or a finer-grained one.

- On case studies the adequacy measure simply captures whether the learned model is *the same* as the target model or not, in terms of behavioral equivalence (see Definition 2.9).
- On random datasets an *accuracy* measure will be used; such measure will range from 0.0 to 1.0 dependent on whether the learned model is considered far or close to the target model. This will be estimated through test samples (see Section 5.3.1).

Note that adequacy is harder to assess on real-world case studies where the target model is unknown. In practice, human inspection of the learned models is required.

Number of scenario questions The number of queries generated to the oracle is a key measure for the usability of QSM in practice.

This is certainly true when the oracle is a human being. However, a large number of queries might also become a problem with automated oracles; indeed, online oracles may be slow, automated oracles might be expensive, etc.

Induction time The time taken to infer a model also deserves special attention. While a reasonable induction time is desirable in any case, fast, real-time interactions are required for usability of QSM by a human oracle.

Our experiments were designed to isolate the effect of the orthogonal features of our inductive technique on the three measures above. Roughly, they quantify the gains and costs of the following ones:

- The use of an oracle who can answer scenario questions: a gain is expected in model adequacy at the cost of a longer induction time.
- The use of the Blue-fringe heuristic instead of the RPNI search order: a gain in adequacy is expected as well as a reduction of the number of scenario questions;
- The use of domain knowledge such as fluent and goals: a gain in adequacy and a reduction of scenario questions should be observed as well;
- The use of control information encoded into a hMSC: here also, a gain in adequacy is expected.

5.2 Evaluations on case studies

QSM and ASM were evaluated on three different case studies of varying complexity. The first one is a mine pump system inspired from [Jos96] and often used as a "benchmark" in the literature. The second case study refers to an extended version of the train system used here as a running example. The third one is a phone system handling communications between a caller and a callee.

Section 5.2.1 details the evaluation methodology. Section 5.2.2 illustrates a typical modeling session in the ISIS tool on the mine pump system. The subsequent sections then further discuss the impact observed on model adequacy, number of scenario questions and induction time of the different induction features, such as using an oracle or the Blue-fringe heuristics.

5.2.1 Evaluation methodology

Beyond a practical evaluation of our tool-supported approach to multi-view LTS synthesis, our objective was to assess the impact of constraining induction through fluents, models of external components, domain descriptions and goals. Such impact was measured in terms of the number of generated scenario questions and the adequacy of the synthesized models. Induction time was also measured so as to check that fast interactions with the oracle are observed.

For each case study, we proceeded in two steps:

1. (a) Design a scenario collection allowing for meaningful subsequent comparison, that is, sufficiently rich to allow an adequate system LTS to be induced under one setting of the experiment at least (see hereafter).

   (b) Define a common set of fluent definitions identifiable from this scenario collection. From this set of fluents, define a common set of domain properties and goals.

2. Evaluate the techniques on this scenario collection, without and then with fluents, goals, domain descriptions, or models of external components.

In Step 1, the richness condition on the scenario collection (a) amounts to require the collection to be structurally complete; every transition in the target system LTS must occur in at least one scenario (see Section 4.2). The ISIS tool was used to incrementally set up such a scenario collection (see Section 7.2):

- An initial set of scenarios that end-users would typically provide was first selected.

- By generating scenario questions, adding domain properties and goals, and validating the induced LTS, additional scenarios were found that were initially missing.

- Some of these scenarios were added to the collection for the comparisons in Step 2. Added scenarios have been selected so as to reach a structurally complete collection of scenarios without making it too rich.

Problem	Events	States	Trans.	$\|S^+\|$	$\|S^-\|$	Avg. length
Mine Pump	8	10	13	3	0	8
Train	13	17	23	3	0	9
Phone	16	23	33	6	4	11

Table 5.1: Sizes of the case studies.

The size of the scenario collection resulting from Step 1 is shown in Table 5.1. $|S^+|$ and $|S^-|$ denote the number of positive and negative scenarios, respectively. The average scenario length is reported in the last column. The size of the target system LTS are also reported in terms of number of different event labels (alphabet size), states, and transitions.

In order to measure the number of generated scenario questions in Step 2, an oracle was implemented to simulate the end-user. This oracle knows the system LTS for each problem and correctly classifies generated scenario as positive or negative. Note that this is a strong assumption on the oracle, especially when played by an end-user in practice; this issue has been discussed in Section 4.7.

The next section illustrates how the ISIS tool were used to build the scenarios, domain properties and goals for the Mine Pump system. Then, results of evaluation experiments will be detailed.

5.2.2 Modeling the Mine Pump system with the ISIS tool

This section shows the ISIS tool in action on a classical Software Engineering benchmark. We consider the following simplified problem statement for the Mine Pump exemplar [Jos96]:

> *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.*

The tool allows specifying positive and negative scenarios. In addition, if available, other kinds of knowledge can be captured. Such information includes fluents, state machines of legacy components or safety properties to be met by the system.

Fig. 5.1 shows two positive scenarios initially drawn from the description above. As stated in Chapter 4, our inductive technique make the built-in

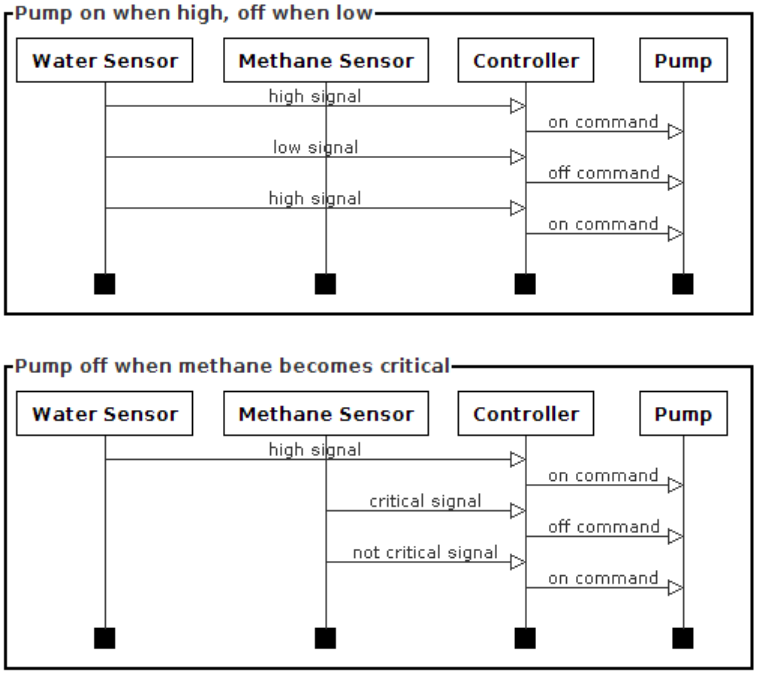


Figure 5.1: Initial scenarios for the Mine Pump system.

assumption that such input scenarios start in the same initial state. The submitted scenarios thus implicitly specify that the water level is low and the methane level is below the critical level.

The first positive scenario illustrates that the pump should be turned on by the controller when the water level is high and turned off when the water level is low. If the water gets high once again, the controller simply turns the pump on in response.

The second scenario shows that the pump should be turned off when the methane level becomes critical. As the water is high, the pump is switched back on when the methane level becomes non critical.

Running QSM with these two scenarios

QSM is ran a first time with these two scenarios as input. For our case-study evaluations, QSM was actually used with the Blue-Fringe heuristic and a consolidation threshold of 1 (see Section 4.3.3). This means that two states of the PTA are considered for merging only if they share at least one outgoing event. Such consolidation threshold drastically reduces the number of possible scenario questions; generated questions also tend to be guided by more evidence, which fits the intuition. In our experience, doing so results

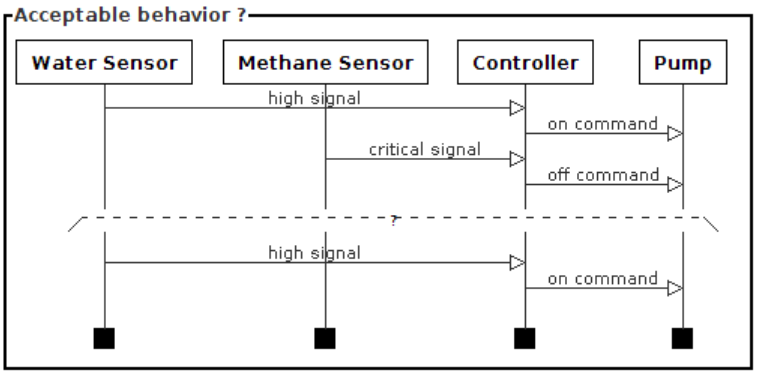


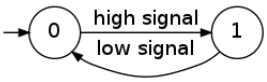
Figure 5.2: A first scenario question submitted by QSM on the Mine Pump case study.

in a conservative state-merging approach that is well suited for building a new specification from scratch.

With such a configuration, only one scenario question is submitted for classification. It is illustrated in Fig. 5.2 and can be rephrased as: "*after having switched the pump off because of a critical level of methane, may the controller switch the pump on if the water becomes high once again?*". At least two reasons support the rejection of such a scenario:

- A domain property prevents the water from becoming high twice in a row. Observing two "high" signals from the water sensor consecutively brings our attention to the fact that the scenario should probably be rejected.

  This domain property might be formally captured in two ways: either by a legacy LTS for the water sensor (see Fig. 5.2.2) or by a declarative domain property.



  If the declarative option is chosen, the following LTL safety property can be used to capture the fact that the "high water" signal is only observed provided that the current water level is low:

  $$\text{DomProp[Water]} = \circ(\text{high signal}) \Rightarrow \neg\text{HighWater}$$

  $$\text{fluent HighWater} = \langle\{\text{high signal}\}, \{\text{low signal}\}\rangle \text{initially } \mathit{false}$$

- In addition, the level of methane is still critical in the second part of the scenario question. In such a situation the controller may not switch the pump on.

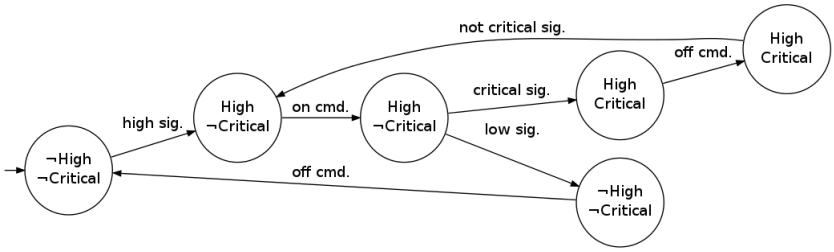


Figure 5.3: The annotated LTS of the Pump Controller. Here "sig." stands for "signal" and "cmd." stands for "command", "High" stands for "High-Water" and "Critical" for "Critical Methane".

This second rejection reason might be captured through the following safety goal.

Avoid [PumpOnWhen Methane] =
CriticalMethane $\Rightarrow \circ(\neg$PumpOn)
fluent CriticalMethane =
<{critical signal}, {not critical signal}>initially *false*
fluent PumpOn =
<{on command}, {off command}>initially *false*

Note that the scenario might be rejected without further explanation. As illustrated above, the rejection of a scenario is a rich opportunity to declare state variables and identify requirements and domain properties. However such more formal steps are optional and may simply be bypassed by non-experts. Not specifying fluents, domain properties and goals typically lead to more scenario questions.

In our modeling session, only the `CriticalMethane`, `HighWater` fluents above were defined. No further scenario questions were submitted for classification. The resulting LTS for the pump controller agent is illustrated in Fig. 5.3.

A second modeling step

At this step various choices can be made:

- More positive and negative scenarios can be added to the initial scenario collection. QSM is then ran once again and generates new scenario questions.
- The consolidation threshold of Blue-fringe can be lowered. This results in a more aggressive state merging approach, leading to new scenarios

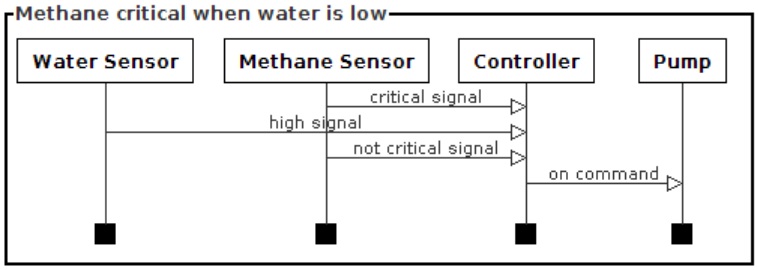


Figure 5.4: The methane level may become high when the water is low.

to classify as positive or negative system behaviors. Doing so triggers for further identification of domain properties and goals.

- Synthesized state machines may be decorated with fluent invariants and inspected by experts. New positive scenarios are typically added to the scenario collection so as to cover missing behaviors. New negative scenarios can be added when over-generalization is detected.
- Additional synthesis and/or analysis tools can be launched on the specification, such as an animator or a model checker. In particular, the ISIS tool supports the inference of goals from scenarios [Dam06, Dam11].

The last option was chosen in our case, the ISIS tool being requested to infer maintain goals from the annotated scenarios. Only one property was inferred and submitted for classification:

$$\Box(\text{HighWater} \vee \neg\text{CriticalMethane})$$

This property states that either the water is high or the methane is below the critical level. It has to be rejected as those two phenomena are independent of each other. A counterexample scenario must illustrate a situation where the water level is low while the level methane is critical:

$$\Box(\neg\text{HighWater} \wedge \text{CriticalMethane})$$

Fig. 5.4 provides such counterexample where the "critical signal" event is observed in the initial state. The pump may then be switched on only if the methane level is below the critical level.

Running QSM a second time

As a new scenario is available, QSM is ran so as to obtain revised agent state machines. With a consolidation threshold of 1, no scenario question

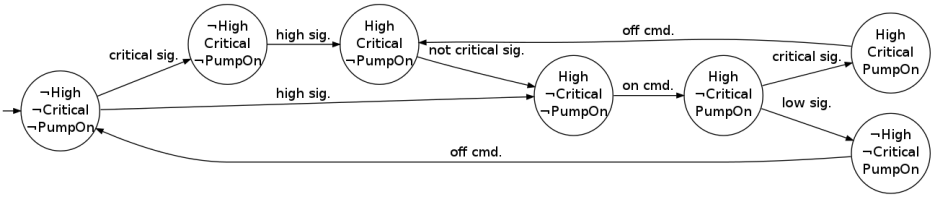


Figure 5.5: A second version of the Pump Controller LTS

is submitted for classification. The revised LTS state machine of the pump controller is shown in Fig. 5.5.

A manual inspection of this state machine showed that only one transition was missing, that would allow the methane to become critical then not critical from the initial state. This was fixed by simply adding such a loop at the beginning of the third scenario and running QSM once again (see Fig. 5.6).

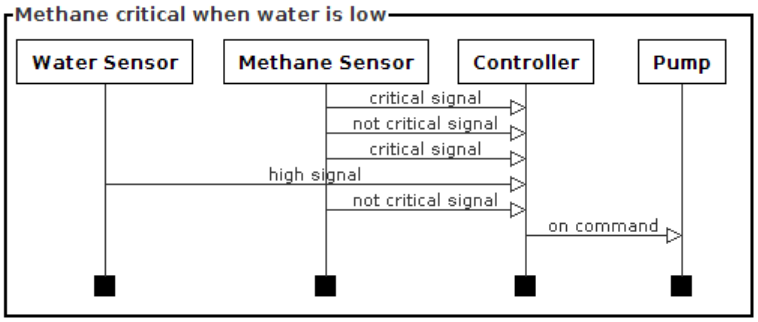


Figure 5.6: Third scenario revised to include a loop through the methane events.

Inferring goals

From fluent definitions and the four available scenarios (3 positive and 1 negative), the tool was requested to infer goals. Three of them were kept as typical requirements of the mine pump system:

- Maintain[PumpOn When HighWater and Not Critical Methane]

$$\text{HighWater} \wedge \neg\text{CriticalMethane} \Rightarrow \circ(\text{PumpOn})$$

- Avoid[PumpOn When LowWater]

$$\neg\text{HighWater} \Rightarrow \circ(\neg\text{PumpOn})$$

- Avoid[PumpOn When CriticalMethane]

$$\text{CriticalMethane} \Rightarrow \circ(\neg \text{PumpOn})$$

These goals were used to prune the induction in controlled experiments. The resulting specification was considered rich enough; it enabled a deeper evaluation of our inductive algorithm by controlling various parameters such as the number of injected goals. A similar process were used on the two other case studies. The results are described in the next sections.

5.2.3 RPNI search order vs. Blue-fringe strategy

Table 5.2 reports the results obtained when running QSM with the RPNI search order and the Blue-fringe heuristics, respectively. In the sequel, these two settings will be denoted by QSM-rpni and QSM-fringe, respectively. The comparisons are made without additional knowledge to constrain induction.

Problem	Algorithm	Q^+	Q^-	Model adequacy
Mine Pump	QSM-rpni	1	30	missing/unallowed traces
	QSM-fringe	1	4	adequate model
Train	QSM-rpni	4	83	adequate model
	QSM-fringe	5	5	adequate model
Phone	QSM-rpni	5	171	missing/unallowed traces
	QSM-fringe	5	19	missing/unallowed traces

Table 5.2: RPNI search order versus Blue-Fringe strategy for QSM.

The table shows the number of queries the oracle had to answer together with a binary measure of model adequacy. Q^+ and Q^- denote the number of accepted and rejected scenario questions, respectively. The model is said to be *adequate* if it matches the known target LTS, formally if the two models are trace equivalent (see Definition 2.9).

The following main observations can be made from these results:

- The large number of generated questions makes QSM-rpni unusable with end-users on bigger systems. In contrast, the number of rejected scenario questions is drastically reduced thanks to the Blue-fringe search strategy.
- In the phone system, an adequate system LTS cannot be synthesized with the sole use of the initial scenario collection and scenario questions. Wrong generalizations do occur; some states are merged whereas they need to be distinguished. A richer scenario collection would allow to correctly identify the target model; remember, however, that

the initial collections have been chosen so as to observe the gain in adequacy when injecting fluent definitions and goals in the synthesis process.

- 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 be restarted when an incorrect search path has been taken (see Algorithm 1 in Section 4.3).

As the Blue-fringe heuristics appeared by far superior to the RPNI search order, subsequent comparisons were made only with QSM-fringe.

5.2.4 Impact of fluent propagation

In this second evaluation, the synthesis is performed for each case study on an increasing number of fluents among those available.

Problem	Nb. fluents	Q^+	Q^-	Model adequacy
Mine Pump	0	1	4	adequate model
	1	1	1	adequate model
	2	1	0	adequate model
	3	1	0	adequate model
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	0	5	19	missing/unallowed traces
	1	5	13	missing/unallowed traces
	2	6	9	adequate model
	3	6	4	adequate model

Table 5.3: Impact of fluent propagation.

Table 5.3 summarizes the influence of fluent decorations to constrain the induction process. The following observations can be made:

- The number of rejected scenario questions is decreasing as the number of fluents is increasing. Such questions can even disappear when the set of fluent definitions is large enough.

- In contrast, for the same induced LTS, the number of accepted scenarios remains the same. Fluent-based state information can only increase the number of incompatible states and hence, reduce the number of rejected scenarios.
- As seen in the phone system, fluent definitions yield a better model adequacy. Together with the initial scenario collection and the answers to scenario questions, two fluents were sufficient for a trace equivalent model to be found.

5.2.5 Impact of goals and domain properties

From a scenario collection and fluent definitions, the ISIS tool can automatically infer a variety of requirements and domain properties using an inference technique described in [Dam06, Dam11] (see Section 7.2). This feature was used to measure the impact of goals and domain properties on the induction process.

For the Mine Pump system, three important 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.*

Those inferred properties were used in turn to incrementally constrain the induction process. Table 5.4 shows the results obtained.

Compared with fluent injection, similar observations can be made, that is, goals help reaching a better model adequacy while reducing the number of rejected scenario questions. Goals and domain properties are however seen to be slightly 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.

Problem	Nb. properties	Q^+	Q^-	Model adequacy
Mine Pump	0	1	4	adequate model
	1	1	0	adequate model
	2	1	0	adequate model
	3	1	0	adequate model
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	0	5	19	missing/unallowed paths
	1	6	6	adequate model
	2	6	4	adequate model
	3	6	4	adequate model

Table 5.4: Impact of inferred properties on induction.

5.2.6 Combined use of fluents, properties and external components

Table 5.5 shows the results of QSM-fringe induction constrained with all fluents, goals and domain properties, and foreign component(s) available in each case study. (The fact that the number of fluents and goals is the same for each case study is purely coincidental.)

For each problem, the first line corresponds to the simplest approach, that is, QSM with the RPNI search order and no domain knowledge. The second line shows how much is gained when the various techniques are combined to constrain the interactive synthesis process. For example, QSM-fringe inferred an adequate model for the phone system with only 3 rejected scenario questions; in contrast, 172 rejected questions were not sufficient to find such model with QSM-rpni.

Problem	Search	Fl.	Goals	Comp.	Q^+	Q^-	Adequacy
Mine Pump	QSM-rpni	0	0	0	1	30	not adequate
	QSM-fringe	3	3	2	1	0	adequate
Train	QSM-rpni	0	0	0	4	83	adequate
	QSM-fringe	5	5	2	5	0	adequate
Phone	QSM-rpni	0	0	0	5	172	not adequate
	QSM-fringe	3	3	1	7	3	adequate

Table 5.5: Combining fluents, properties and external components to constrain induction.

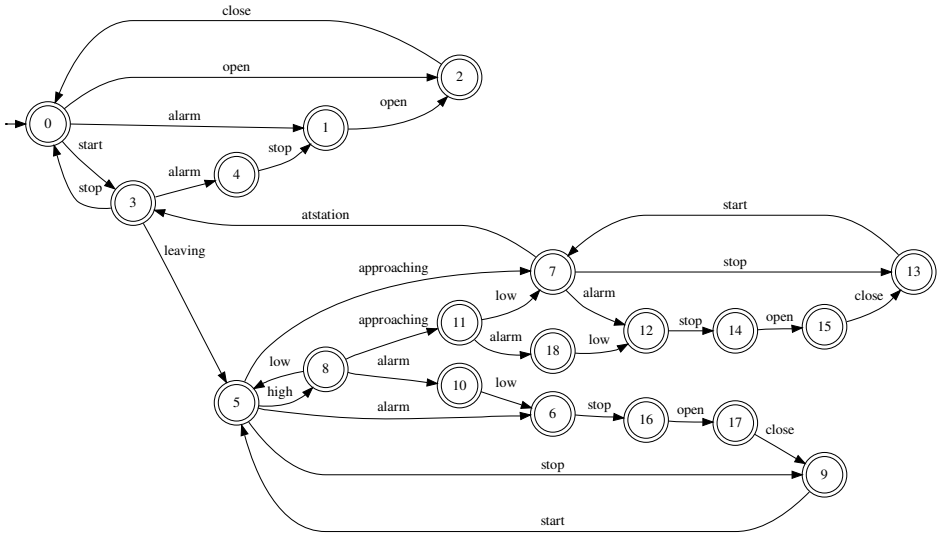


Figure 5.7: Target model of the train system.

5.2.7 Impact of using additional control information

To measure the impact of using a hMSC as input of the synthesis process, ASM has been evaluated on an extended version of our train system. The evaluation protocol is slightly different from the one presented in the previous section.

- The target model of the train system was first been built using ISIS, yielding a system LTS of 19 states and 10 events (see Figure 5.7).
- A typical collection of scenarios was also built for the system and represented as an augmented PTA, yielding 9 positive and 5 negative scenarios for a total of 55 states.
- Using the target LTS, a few state pairs of the PTA were identified as corresponding to the same system state, and immediately merged. This early state merging occurs before launching the inductive algorithm. The automaton resulting from this phase was then given as input to ASM; its performance was then evaluated (see below). The algorithm generalizes the automaton by merging additional state pairs under the control of the negative scenarios.

The early state merging phase simulates the control information typically offered by a hMSC. State pairs of the PTA that are merged early have been chosen following a "loop identification" heuristic, representative of the way such a specification is incrementally built by an end-user using an hMSC.

For instance, opening and then closing the doors from the initial state naturally returns to the initial state (see the loop between states 0 and 2 in Figure 5.7). Some loops are less obvious to identify from the scenarios, e.g., the sequence of events (*leaving*, *high*, *approaching*, *low*, *atstation*) forms a loop starting from state 3.

Equivalent state pairs identified this way were classified in four categories, according to the expected difficulty for an end-user to discover them in the scenarios. ASM was then evaluated on increasing proportions of such state pairs being merged early: 3%, 6%, 10% and 15%. This simulates the use of an increasingly rich hMSC as input to the induction process.

Algorithm	% equivalent state pairs merged early	Accuracy
RPNI	-	0.55
Blue-fringe	-	0.83
ASM	0 %	0.55
	3 %	0.71
	6 %	0.73
	10 %	0.88
	15 %	0.90

Table 5.6: Classification accuracy obtained with different setups on the train case study.

Table 5.6 compares the accuracy of the LTS learned using RPNI, Blue-fringe and ASM, with an increasing percentage of equivalent PTA state pairs merged early. (Remember that in our current implementation, ASM uses the same search order as RPNI and does not benefit from the Blue-fringe heuristic – see Section 4.5).

As seen in Table 5.6, the reported model adequacy is no longer a binary measure here. Instead, an accuracy measure of the learned model is reported as the average classification rate computed over 10 independent test samples. Each of these samples contains 80 positive or negative scenarios randomly drawn from the target LTS. The classification rate corresponds to the percentage of these scenarios correctly classified by the learned model.

This experiment shows that enriching the input given to ASM leads to better accuracy, which is expected. Interestingly, ASM outperforms Blue-fringe when such control information gets rich enough. In the experiment, it already occurs when 10% of the state pairs known to correspond to the same system state are merged ahead of the induction itself.

The results also show that no algorithm was able to perfectly identify the target model on such sparse samples. In order to isolate the effect of injecting control information in the induction process, only a few negative scenarios were used as source of negative information while other sources do exist, such as fluents and goals.

5.2.8 Induction time

Only the adequacy and the number of generated scenario questions have been discussed so far. The induction time was also systematically monitored in our evaluations as it drives the usability of QSM in practice.

All experiments reported in this section were conducted on a Pentium IV, 1.8 GHz, 512Mb with Java 5.0. When using QSM, our tests showed that the maximum time between two scenario questions was 40ms for the bigger case study. The interactions with the end-user are performed in real-time. No performance problems are expected with respect to user interactivity for typical sizes of requirement models, even if larger than those considered here by an order of magnitude.

5.3 Experiments on synthetic datasets

In addition to experiments on case studies, QSM and ASM in Chapter 4 were also evaluated on synthetic datasets. The aim was to study their performance when the problem size, in terms of number of states of the target LTS, grows significantly beyond those of the case studies.

- This allow illustrating the expected performance of these algorithms on state machines whose size is more representative of real-world cases.
- It also provides performance profiles of the algorithms, for application contexts where the sizes of the target machines and/or of the learning samples are bigger. These profiles are given in terms of accuracy, number of scenario questions and induction time reported for increasing learning sample sizes.

For reasons explained in Section 5.1, however, no additional domain knowledge was used to constrain the induction process.

Section 5.3.1 describes the methodology used to generate automata, learning and test samples. Sections 5.3.2 and 5.3.3 then discuss the evaluation of QSM and ASM, respectively.

5.3.1 Evaluation protocol

The procedure used to evaluate ASM and QSM on synthetic data inspired from the Abbadingo protocol [Lan98]. Roughly, evaluating an induction algorithm on a given target automaton consists in running it on learning samples of increasing size in terms of the number of positive and negative

strings that they contain. The accuracy of an induced model is then measured in terms of the number of strings of an independent test sample that the learned model correctly classifies.

The experiments here were made on random LTS of increasing sizes in the range $n = 20$, 50, 100 and 200 states. Only alphabets of two symbols were considered – a feature inherited from Abbadingo[1]. To match our application context, experiments were performed on LTS instead of the more general class of DFA (see Section 2.3.1). All states of the random automata are thus accepting states.

Inspired from Abbadingo, the randomly generated LTS were trimmed to remove unreachable states, and minimized to obtain canonical target machines (see Section 2.3). Moreover, only automata without terminating state were kept for the experiments. Such states typically capture deadlocks in multi-agent systems and should therefore be avoided.

In order to build learning and test samples for a given target LTS of size n, an initial set of n^2 different strings was first synthesized. These strings were randomly generated without replacement using a uniform distribution over the collection of all binary strings of length $[0, p + 5]$, where p is the depth of the automaton. (The depth of an automaton is defined as the length of the longest shortest path from the initial state to any other state.) This bound was chosen so as to ensure that deepest states have a good chance of being reached by at least one input string of a sample. This is a necessary condition for structural completeness of the sample (see Section 4.2), which was not guaranteed however.

Our procedure for generating samples ensures them to contain positive and negative strings in roughly equal proportion. Roughly, this is a consequence of the kind of automata considered.

A maximal sample size of $\frac{n^2}{2}$ strings was experimentally observed as offering the convergence for all tested algorithms (RPNI, Blue-fringe, QSM-rpni, QSM-fringe and ASM). The learning experiments were conducted on increasing proportions of this nominal training sample, i.e. 3%, 6%, 12.5%, 25%, 50% and 100%.

Test samples of at most $\frac{n^2}{2}$ strings were used for measuring the generalization accuracy of the learned model. The accuracy measure here is defined as the percentage of test strings correctly classified by the learned model.

Training and test samples were designed so as to not overlap. To conserve their independent nature, the test samples did not contain the additional strings that were submitted to the oracle during a QSM learning phase.

[1]See Chapter 6 for additional experiments on larger alphabets and a study of the influence of the alphabet size on induction algorithms.

All experiments reported hereafter were performed on at least 10 randomly generated LTS for each size and 5 randomly generated samples for each of them.

5.3.2 Evaluation of QSM on synthetic datasets

This section discusses evaluation results for the QSM algorithm. An automatic oracle was implemented to answer the questions asked during its execution. This oracle correctly answers the scenario questions since it has access to the target LTS in such controlled experiments.

Generalization accuracy

Figure 5.8 reports the proportion of independent test samples correctly classified for several target sizes. This accuracy measure is reported while increasing the learning sample size. Comparative performance data are given for RPNI, Blue-fringe, QSM-rpni (QSM with the RPNI merging order) and QSM-fringe (QSM with the Blue-fringe strategy).

Results from Section 5.2.3 are confirmed here; Blue-fringe (resp. QSM-fringe) outperforms RPNI (resp. QSM-rpni) for sparse training samples. Moreover, significant improvements in generalization accuracy is observed thanks to the interactive feature.

- The QSM algorithm outperforms the original RPNI and Blue-fringe systematically.
- Interestingly, QSM-rpni also overcomes the original Blue-fringe algorithm. In terms of model accuracy and for a fixed learning sample size, the interactive feature of QSM is at least as powerful as the evidence-driven heuristic search of Blue-fringe.

Observe that the learning sample sizes were well chosen to illustrate the convergence of this family of induction algorithms:

- For a given target size, the curves show the different induction phases. When the sample is sparse, 3% or 6% for example, poor accuracy results are first observed. The accuracy grows while the learning sample becomes larger; it does so rapidly until an accuracy of about 0.95 is reached. Beyond this point, the accuracy continues to grow with the learning sample size but much slowly.
- The relative performance data described above do not depend much on the target size. The curves seem successively shifted left when the target size grows. This can be explained by the fact that a quadratic

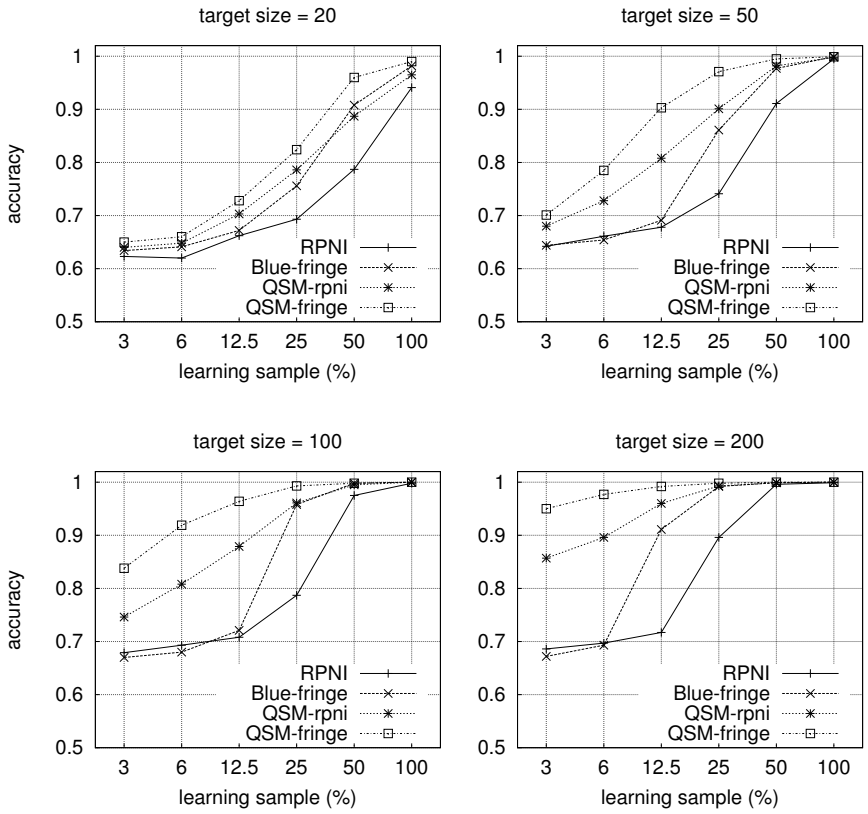


Figure 5.8: Classification accuracy of QSM.

learning sample in the size of the target LTS tends to be richer for large LTS than for smaller ones.

Number of scenario questions

The number of scenario questions generated to the oracle is an important feature of QSM. When the oracle is an end-user, this factor heavily impacts on the practical usability of the approach.

Figure 5.9 shows the number of generated questions depending on the learning sample size, for several target sizes. The results are given for QSM-rpni and QSM-fringe. In each case, the number of generated scenarios classified by the oracle either as positive or negative are reported separately.

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

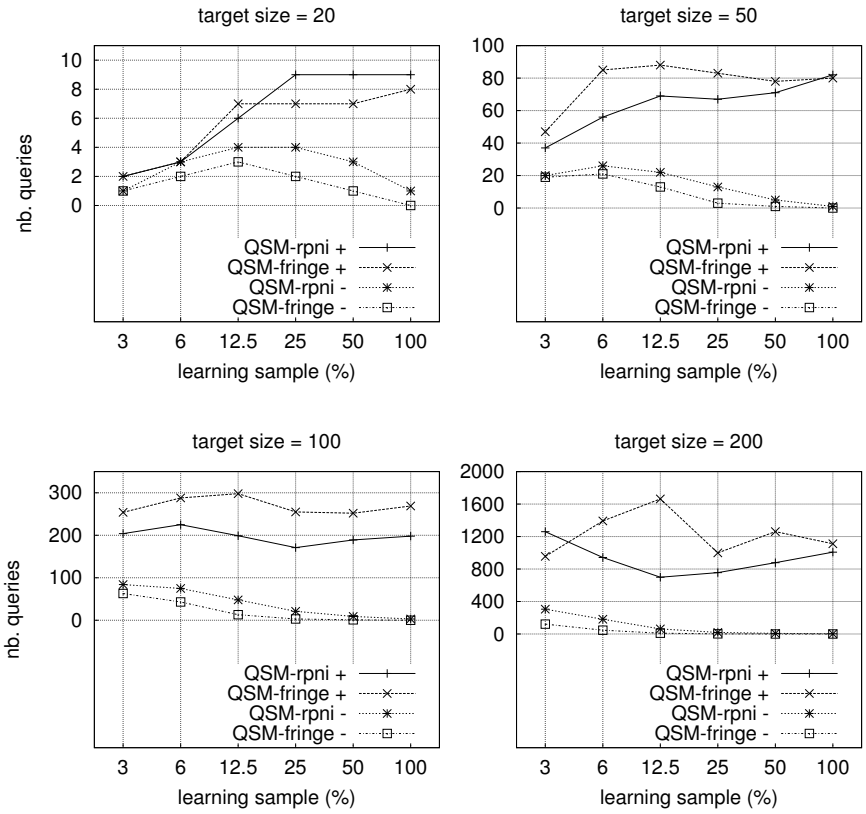


Figure 5.9: Number of scenario questions generated by QSM.

the negative strings.

- QSM-fringe tends to generate fewer strings classified as negative by the oracle than QSM-rpni. In both cases, however, the number of rejected queries decreases with the learning sample size. As a comparison of Fig. 5.9 and Fig 5.8 shows, it even reaches zero when the algorithm has converged to the exact model.

Induction time

Figure 5.10 shows the induction time for varying learning sample size and target size. All tests were executed with Java 5.0 on a Pentium-IV 3 GHz computer with 1Gb of RAM.

As seen there, the RPNI, Blue-fringe and QSM algorithms go through different phases according to the amount of data available:

- Initially, CPU time tends to increase with the learning sample size. In

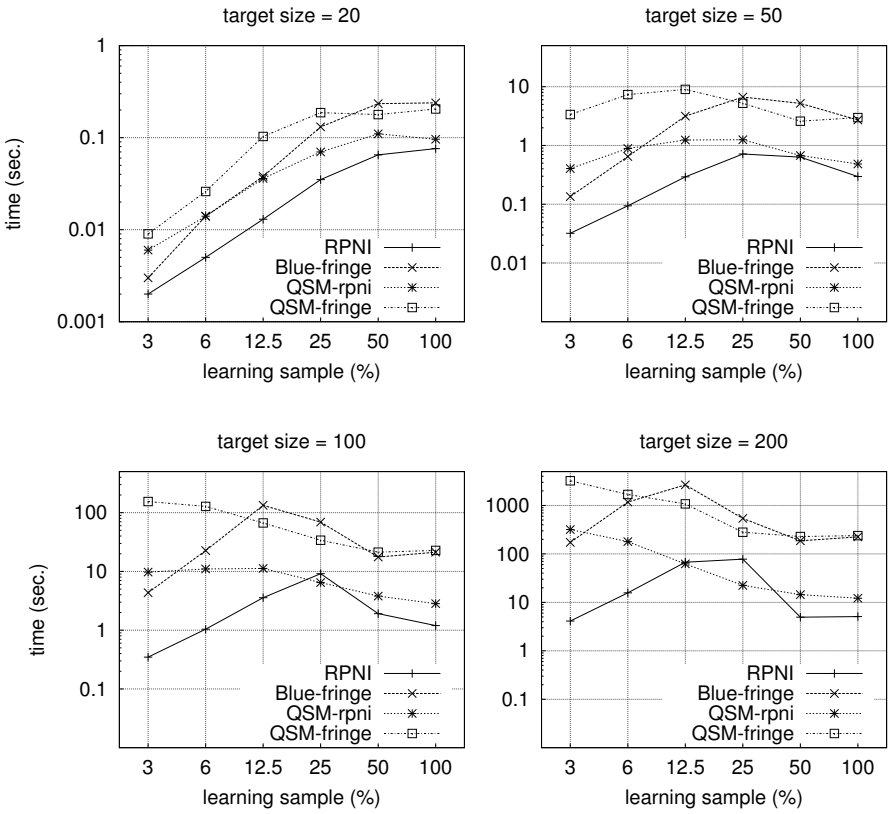


Figure 5.10: Induction time with QSM.

this first phase, the learning time follows the increase of the sample size.

- When the learning sample becomes richer, better generalizations can be obtained by merging states in a more sound way. A comparison between curves in Fig. 5.8 and Fig. 5.10 reveals that the classification rates of new data increases while the learning time is reaching its maximum.

- The last phase is observed when the algorithm rapidly converges to a good model. Classification accuracy tends towards 100% while CPU time is decreased because the right merging operations are performed directly.

The known tendencies of induction algorithms in the RPNI family were confirmed by our experiments (see, e.g., [Lan98]). However, the curves of QSM are shifted left with respect to the learning sample size. As already observed in Fig. 5.8, convergence is indeed faster for the QSM algorithm in

that it occurs on sparser samples than RPNI or Blue-fringe. Compared to these, the relative time performance of the QSM algorithm actually depends on two contradictory effects:

- On the one hand, whenever a string is classified as negative by the oracle, QSM is called recursively on an extended sample. Each new call increases CPU time; such call could be considered as a new run of the RPNI or Blue-fringe algorithm. This run can however be interrupted and replaced by another one if a new negative example is included after an additional query.
- On the other hand, due to its faster convergence QSM can obtain better results with fewer data originally provided.

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

5.3.3 Evaluation of ASM on synthetic datasets

The evaluation of ASM on synthetic data is very similar to the one conducted on case studies in Section 5.2.7. The objective is to quantify the gain in generalization accuracy that can be expected when the proportion of domain-specific control information à la hMSC is increased as input.

The experimentation protocol used to achieve this objective is a slight adaptation of the one discussed in Section 5.3.1.

- Experiments here were made on randomly generated target LTS with 32 and 64 states. Following Abbadingo, alphabets still had two symbols only (we further discuss this limitation in Chapter 6).
- Learning and test samples were randomly generated as before. RPNI and Blue-fringe were both evaluated on these samples to provide a reference for the comparisons with ASM.
- In order to provide its input to ASM, a specific procedure simulated the control information given by a hMSC:
  - For a given learning sample, an augmented PTA was first built. As in Section 5.2.7, an early state merging phase then occurred.

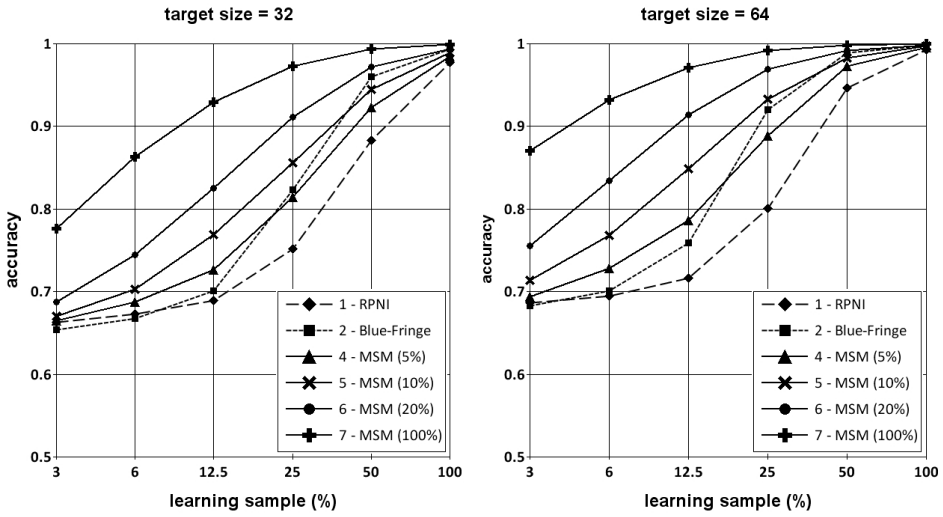


Figure 5.11: Classification accuracy for ASM.

The idea was to merge specific state pairs of the PTA that were known to correspond to the same state in the target LTS.

– For this, unique labels were associated to randomly chosen states of the target LTS. Increasing proportions of the number of states labeled in this way were used, namely, 5%, 10%, 20% and 100%. The PTA and the target LTS are were then jointly visited and state labels are reported as decorations of the PTA states.

– States of the PTA sharing the same label were merged early. This effectively simulated the availability of control information through a hMSC by introducing loops in the input scenarios. The automaton resulting from this step was then used as input to ASM, that further generalized it by merging additional state pairs under the control of the negative strings.

Figure 5.11 reports the proportion of independent test samples correctly classified while increasing the learning sample. Curves in these plots correspond to executions of RPNI, Blue-fringe and ASM with different labeling proportions. Each point in these plots represents the average value computed over 200 independent runs.

ASM overcomes RPNI on all executions, which is actually expected. Remember from Section 4.5.3 that ASM reduces to RPNI in the special case where its input is a PTA, that is, when no early state merging occurs in our experiment. By construction of our experiment, the early state merging performed are sound. Therefore, they could not hurt the generalization

accuracy; hence, ASM can only produce the same or a better accuracy score than RPNI.

From the standpoint of generalization accuracy, however, 5% of the labeling information is comparable to the use of the Blue-fringe heuristic for selecting state pairs to be merged. Beyond this proportion, the accuracy keeps increasing. Interestingly, the accuracy gain is already visible when the sample is sparse.

Note that the identification of the target does not reduce to a trivial problem even with 100% of labeling information. As discussed in Section 4.5, control information is complementary but does not substitute to the negative knowledge provided by negative scenarios, fluent and goals. In particular, it does not prevent overgeneralization from occurring when incompatible state pairs are merged.

5.4 Discussion

This chapter discussed our evaluation of the inductive LTS synthesis technique presented in Chapter 4. The evaluations were conducted on case studies complemented with experiments on synthetic datasets. For recall, with respect to the thesis objectives the main evaluation question were:

> *How well does inductive synthesis help building* adequate*,* complete*,* consistent *and* precise *models for the target system considered?*

The following conclusions can be drawn.

- The overal effectiveness of our multi-view approach has been demonstrated on case-studies. Within a few iterations in the ISIS tool, adequate state machines are found. Scenario questions complete the scenario collection with interesting examples and counterexamples of system behavior. They also trigger the identification of important system requirements and domain properties, thereby completing the overal multi-modeling in a consistent way.
- Together with the Blue-fringe heuristics, the interactive feature of QSM is very effective in reaching a good adequacy of synthesized models. The cost in terms of number of scenario questions may be significant, especially for large systems. Therefore, techniques for reducing the number of scenarios to be classified are required in practice.
- Among our pruning techniques, the domain knowledge provided by fluents, goals and legacy components proves very effective. In addition to

reducing the number of scenario questions, such knowledge effectively guides the induction process towards adequate LTS models.

- The control information provided by a hMSC, simulated in evaluations through a dedicated procedure, also results in important gain in adequacy. While guiding the generalization process towards good generalizations, it has been shown that it does not provide the negative knowledge required to avoid overgeneralization.

- When considering model adequacy, induction time and real-time usability, the approach has been shown to scale. Experiments on synthetic data have been conducted on target machines whose size is significantly larger that the case-studies considered. A convergence towards adequate models within reasonable time bounds has been observed in all cases.

- The most important issue with QSM is related to the number of generated scenario questions. As shown in Table 5.5 and in Fig. 5.9, the number of rejected questions tends to decrease as the induction input gets richer, either because the sample gets larger or thanks to available domain knowledge. This contrasts with scenarios to be accepted, whose number tends to become constant in that case.

  This means in practice that QSM will generate scenarios even when its input is rich enough to guarantee very good generalizations without even asking questions – e.g., when given a characteristic sample. Moreover, fluents and goals do not help reducing these questions. As their number might become large for non-toy systems, improvements of our technique might be necessary for guaranteeing the scalability of the approach.

  One possible solution could be to extend the possible answers from the oracle. For example, the end-user could request the algorithm to terminate immediately, that is, without generating further scenario questions. The inferred state machine would then be submitted for inspection, leading to a form of equivalence queries. When the state machine is not adequate, such queries are typically answered with an additional negative scenario and the induction restarted – e.g., as in L^* [Ang87] (see also Chapter 8).

Let us close this chapter with specific issues related to the experiments themselves and the protocols used to conduct them.

- The coverage of conducted experiments could be slightly extended. In particular, ASM could be further studied on more case studies. The real use of a hMSC instead of a simulation procedure for control information is worth considering as well.

Moreover, our current implementation of ASM relies on the RPNI search order and does not support the interactive feature of QSM. This limited the possible comparisons during evaluations. In particular, the effect of control information on the number of generated questions has not been studied.

- The experiments on synthetic datasets provide performance comparisons for RPNI, Blue-fringe, QSM and ASM. The former two were used as state-of-the-art induction algorithms.

  Another induction algorithm called DFASAT recently appeared that significantly outperforms RPNI and Blue-fringe [Heu10] (see Chapter 6). This opens new perspectives for comparisons, evaluations and further developments around QSM and ASM.

- In contrast to the smaller but "real" case studies, the experiments on synthetic datasets referred to target LTS defined on alphabets of two symbols only. This is a consequence of reusing the protocol from Abbadingo for conducting experiments.

  As shown in the case studies themselves, state machine models of software systems are commonly defined on much larger alphabets. While this does not hurt the soundness of our evaluations, it slightly prevent us from generalizing their results. The next chapter will address this issue by describing further work on our Stamina platform for evaluating inductive model synthesis techniques.

Chapter 6

Towards an Evaluation Platform for Inductive Model Synthesis

Grammar induction algorithms are commonly evaluated by reporting accuracy measures on the learned model for increasing sizes of the training sample. For example, plots in the previous chapter illustrated the convergence of QSM and ASM towards a "good" model when their input becomes rich enough.

This kind of setup can also be used to compare various induction techniques by evaluating them on common benchmarks of increasing difficulty. Along this line, the Abbadingo contest had a notable impact [Lan98]. A grid of grammar induction challenges unbroken at that time were proposed online. The competition resulted in the discovery of the Blue-Fringe heuristics used by QSM for selecting state pairs to merge (see Section 4.3.3). Since then, Abbadingo has been frequently used as a reusable benchmark and evaluation protocol for grammar induction, e.g., [Luc03, Bon05, Luc05, Adr06, Dup08, Lam08, Heu10].

A serious weakness of the Abbadingo benchmark is its restriction to induction problems for automata defined on alphabets of two symbols only. Freezing certain parameters, such as the alphabet size, is necessary for keeping a competition accessible to participants. As a side effect, it limits the generalization of the conclusions and the reusability of the competition protocol.

This effect has been observed when conducting the evaluations reported in the thesis. On one side, Abbadingo allowed our results to be compared on a sound and agreed benchmarking protocol. On the other side, limiting the evaluations to binary alphabets lacks credibility from a software

engineering standpoint. Behavior models are commonly defined on thirty events or more. One might thus reasonably question the representativeness of synthetic machines used in the previous chapter for capturing behavior models.

Extending the Abbadingo protocol itself to take larger alphabets into account appears inappropriate. By design, the generation of learning samples in Abbadingo is independent from the target automaton. This feature inspires from *probably approximately correct* (PAC) learning, a sound theoretical framework from machine learning [Val84]. However, this independence assumption between that target automaton and the learning sample does not accurately reflect the way samples are obtained from end-user scenarios or automated testing of distributed systems. In particular, generating strings with a distribution independent of the state machine would lead to an overwhelming amount of negative scenarios for typical behavior models. Abbadingo is further discussed in Section 6.1.

For that reason, an alternative benchmark is proposed here, called Stamina (for *State Machine Inference Approaches*). Stamina inspires from Abbadingo but focuses on the complexity of the learning with respect to the alphabet size. Our protocol relies on adapted procedures for generating automata and samples.

Stamina has initially taken the form of an induction competition, in a similar spirit to Abbadingo. The competition has officially ran between March and December 2010 and was organized in collaboration with the universities of Sheffield and Leicester; it has been officially sponsored by the Engineering and Physical Sciences Research Council[1]. The detailed setup of Stamina is explained in section 6.2.

The winning algorithm, DFASAT, as well as a few additional competitors have significantly outperformed the Blue-fringe baseline on a variety of problems. DFASAT mixes SAT solving and state merging and uses a new scoring heuristics that proved especially useful to infer the kind of state machines considered in the competition. Section 6.3 presents the main results of the competition and a brief description of the winning technique.

Further to fostering efforts around the induction problem, the competition raised interest from at least three communities, namely, Machine Learning, Software Engineering and Formal Methods. The Stamina website[2] is still accessible; it has been updated to serve as an online benchmark and evaluation platform for model synthesis, as discussed in Section 6.4.

[1] http://www.epsrc.ac.uk/
[2] http://stamina.chefbe.net/

6.1 The Abbadingo benchmark

Abbadingo is designed as a grid of 16 induction problems [Lan98]. A competing learner gets a set of training strings labeled as positive or negative by an hidden DFA. It is required to predict the labels that the DFA would assign to a set of unlabeled testing strings. The 16 problems are classified in a grid along two dimensions of difficulty.

- The first is the size of the target automata, as measured by the number of states (64, 128, 256 or 512).
- The second is the sparsity of the training sample, in terms of four sparsity levels. These levels have been tuned by manually inspecting the learning curves of the Trakhtenbrot-Barzdin algorithm [Tra73, Lan92]. This algorithm was a state-of-the-art induction algorithm in 1997; the problem grid has been adjusted in such a way that it solves the four problems with the largest samples.

The target automata, training sets and test strings of Abbadingo were all drawn from uniform random distributions.

- Random automata were generated by constructing and minimizing directed graphs of binary degree. Edge and state labels were chosen by flipping a fair coin.
- A training set for a target automaton A were made of a random sample drawn without replacement from a uniform distribution over the collection of all binary strings. The average length of those strings were chosen in order to have a good chance of reaching the deepest state of the automaton, a necessary criteria toward a structurally complete sample (see Section 4.2.2).
- A testing set were composed of 1800 strings drawn from the remaining strings. Training and test sets did not overlap.

The testing protocol of Abbadingo runs as follows. The learner labels each string of the test set and submits this labeling to an online oracle[3]. To avoid hill climbing, where the feedback of the competition server could be used by the learner to iteratively optimize a first solution, this oracle only provides a single bit of feedback. The latter tells whether or not the problem is broken.

A problem is considered broken if the accuracy of the labeling reached at least 99%. The accuracy is computed as the proportion of the 1800 test

[3]available at http://abbadingo.cs.nuim.ie/

strings correctly labeled. During the competition itself, the first participant to break a given problem gained credit for it. Abbadingo allowed multiple winners by defining a partial order on problem difficulty: a problem A is considered harder than a problem B if its DFA has more states *and* its training sample is sparser.

Two winners, Rodney Price and Hugues Jullié, won the competition with similar algorithms relying on what has since been called *evidence-driven state merging* (EDSM). When performing generalization of a sample through state merging (see Chapter 4), this term captures the strategy of first performing state merges that are supported by the most evidence, according to a specific scoring heuristics. Among other contributions, the competition helped finding the Blue-fringe heuristics to refine which state pairs to merge preferably (see Section 4.3.3).

6.1.1 Limitations for evaluating inductive model synthesis

After the formal competition, Abbadingo evolved into a reusable protocol and online benchmark for induction techniques. The evaluations in Chapter 4 were conducted on a similar protocol. However, Abbadingo sets the size of the alphabet to two symbols only. This limits the relevance of reusing its protocol for evaluating behavior model synthesis techniques. Behavior models are commonly defined on larger sets of events. For instance, the small phone case study studied in the previous chapter uses 16 distinct events for a state machine of only 23 states. Moreover, the automata and samples generated with Abbadingo are notable different from those commonly found in software engineering problems:

Automata The automata randomly generated by Abbadingo have a small, quasi-constant, state degree (in number of incident edges). Being defined on a binary alphabet, a canonical automaton of n states has at least n and at most $2n$ edges; these edges are uniformly distributed over all states.

Automata modeling software systems involve transitions that may be triggered by any of a large number of events. The latter may capture mouse clicks, function names, IO events, etc. The number of outgoing transitions for a given state can be very large and vary significantly from state to state. In the process of designing Stamina, a review of state machines found in the literature has shown the following [Wal08]:

- most states have in- and out-degrees of one or two transitions,
- a few states have a high in-degree, e.g. those modeling exception handling,

- a few states have a high out-degree, e.g. an *idle* state where a software agent waits for external stimuli in terms of input events,
- a few states are *sink* accepting states, that is, they have an out-degree of 0, e.g., explicitly modeling the ability of a system to halt.

The automata generated with Abbadingo do not fit such characteristics, due to the small alphabet size. Moreover, because of the uniform edge distribution implied, the protocol would not naturally lead to automata presenting aforementioned characteristics, even if adapted for handling larger alphabets.

Samples Abbadingo was inspired from the PAC framework for generating samples [Val84]. In this setting the distribution of the observed strings is external to the target machine. Samples are therefore simply drawn from uniform random distribution over the collection of binary strings up to a some prescribed length. The target machine is only used to classify them as positive or negative. The respective number of positive and negative strings is naturally balanced.

This procedure is no longer adequate when working with larger alphabets and state machines presenting the characteristics previously described. Because of the small average out-degree of states, an overwhelming majority of random sequences from Σ^* would be negative strings. Moreover, the few positive strings that would be made available would probably not provide a sufficient coverage of the target machine for obtaining good induction results (see Section 4.2).

For behavior models, a *generative* procedure seems more adequate, where the target machine itself is used to generate the learning sample. This is representative of the way such samples are obtained in the software engineering research: end-users scenarios come from a mental model of the software that is, a variant of the target machine [Com11]; execution traces are sometimes obtained through software testing and monitoring; and so on.

Scoring The accuracy measure is defined in Abbadingo as the proportion of test strings that are correctly classified by the learned automaton. This choice is questionable if one relaxes the assumption of balanced test samples [Wal08]. In the extreme case of a test set largely overwhelmed by negative strings, for example, a learner classifying all strings as negative would still obtain a comfortable score. A better measure should consider the acceptance of positive strings and the rejection of negative ones as equally important.

Conducting sound evaluations requires rethinking important parts of the underlying protocol. To capitalize over and share the cost of such work,

we designed Stamina as both a public complement and an alternative to Abbadingo. Focusing on characteristics of behavior models, it also acts as a call to crossfertilization between the machine learning and software engineering communities. To achieve this, Stamina first took the form of an online induction competition before evolving into an online benchmark, in a similar spirit to Abbadingo. The next section details the Stamina setup as designed for the competition. Changes made to the platform since the end of the competition will be explained in Section 6.4.

6.2 Stamina setup

The competition scenario chosen for Stamina is very similar to the one of Abbadingo.

> *A learner downloads a training set made of positive and negative strings, and induces a model using her induction technique. The learned model is then used to label strings of a test sample; these labels are submitted to the competition server. The latter scores the submission and provides a binary feedback as to whether the problem is considered broken or not.*

While the competition scenario is similar, Stamina differs from Abbadingo in its focus on the complexity of the learning with respect to the alphabet size. It therefore relies on an adapted generation protocol for target automata and samples.

6.2.1 Competition grid

As in Abbadingo, induction problems are classified in a grid. Here, the competition grid is divided into cells of five problems each, where each cell corresponds to a particular combination of sparsity and alphabet size. Table 6.1 shows how the 100 competition problems are distributed over cells. Easier problems (smaller alphabet and larger sample) are toward the upper-left of the table whereas the harder problems (larger alphabet and smaller sample) are toward the bottom-right.

With respect to Abbadingo, the following similarities and differences should be noted:

- An increasing size of the alphabet forms a first difficulty dimension, ranging from 2 to 50 symbols. The lower bound allows result comparisons with Abbadingo on easiest problems; the upper bound is representative of behavior models found in the literature.

	Sparsity			
$\|\Sigma\|$	**100%**	**50%**	**25%**	**12.5%**
2	1-5	6-10	11-15	16-20
5	21-25	26-30	31-35	36-40
10	41-45	46-50	51-55	56-60
20	61-65	66-70	71-75	76-80
50	81-85	86-90	91-95	96-100

Table 6.1: Grid of 100 problems distributing the induction difficulty along two dimensions: sparsity of the learning sample and alphabet size.

- In Abbadingo, the varying automaton size is one difficulty dimension (ranging from 64 to 512). In contrast, Stamina considers automata of roughly 50 states only. These automata provide characteristics of behavior models in terms of the variance of their state degree, among other differences (see Section 6.2.2).

- The second difficulty dimension is the decreasing size of the training sample; it is similar to Abbadingo. Nevertheless, samples in Stamina are generated "from the machine" by a random walk procedure, instead of randomly drawn from all possible strings (see Section 6.2.3).

- Instead of an accuracy measure, submissions in Stamina are scored using a *balanced classification rate* (BCR). BCR places an equal emphasis on the accuracy of an inferred model in terms of acceptance of positive sequences and rejection of negative ones. Obtaining a BCR score of at least 99% is required to consider a problem broken. A cell is broken if its five problems are broken by the same learner (see Section 6.2.4)

- Unlike Abbadingo, Stamina allowed one winner only; the latter is the first learning technique to break a hardest cell among those broken during the competition. To adjust the grid and choose cell difficulties, Blue-fringe has been ran and scored on each problem (see Section 6.2.5). The adjustment consisted in ensuring that Blue-fringe almost breaks the problems of the very first cell (alphabet of 2 symbols and sparsity of 100%)[4] while obtaining a decent score on cells immediately adjacent. An open source implementation of this baseline algorithm has been made available for download.

[4] 3 of the 5 problems in the cell are broken by Blue-fringe, which obtains a BCR score of 98% on the two remaining problems

6.2.2 State Machines

For state machine generation, a quick review of software models has been conducted. Observations were made on a small sample of case study models found in research publications (about 20 systems). State machine models were analyzed in terms of number of states, transitions, alphabet sizes, in-/out degree and depth. Although the sample is too small to form any authoritative conclusion, findings can be interpreted as indicative. Based on these observations, Stamina machines were generated using a variant of the Forest-Fire algorithm [Les07]. The algorithm has been tuned to generate state machines presenting the following characteristics.

Number of states All state machines have approximately 50 states. This is somewhat larger than the conventional state machines identified in the literature; the aim is to ensure that a well-performing technique could scale to infer models for reasonably complex software systems. It has also been decided not to consider state machines of exactly 50 states in order to avoid a strong bias in the benchmark. Automaton sizes actually range from 41 to 59 states. The latter information was not disclosed during the competition.

Accepting state ratio A roughly equal proportion of accepting and rejecting states has been chosen, a similarity to Abbadingo. States of most behavior models, notably LTS, are accepting states. It has however been decided to keep non-accepting states as well. This keeps our setup sufficiently close to former competitions such as Abbadingo; in particular, it avoids restricting the problem to the inference of prefix-closed regular languages (see Section 2.3.1).

Degree distribution Following observations from the literature, state machines exhibit an important variance of their state degree, especially on largest alphabets. They may also have sink accepting states, that is, states with no outgoing transition.

Determinism and minimality Following a common setup of regular inference experiments, all Stamina machines are both deterministic and minimal.

6.2.3 Training and test samples

Training and test samples were generated using a dedicated generation procedure. This procedure aims at simulating the way examples of system behavior are usually obtained in the software engineering community (e.g. a collection of program traces at implementation level, the generation of

scenarios at design level, and so on). Stamina samples present the following characteristics.

Generated by the target A dedicated algorithm were implemented to generate positive strings by walking through the automaton. From the initial state it randomly selects outgoing transitions with a uniform distribution. When an accepting state v is reached, the generation ends with a probability of $1.0/(1 + 2 * outdegree(v))$. This procedure simulates an "end-of-string" transition from state v with half the probability of an existing transition. The length distribution of the strings generated is approximately centered on $5 + depth(automaton)$. As in Abbadingo, this provides a good chance of reaching the deepest state of the automaton. However, no guarantee is given of having a structurally complete sample.

Negative strings Negative strings were generated by randomly perturbing positive strings. Three kinds of edit operation are considered here: substituting, inserting, or deleting a symbol. The editing position is randomly chosen according to a uniform distribution over the string length. Substitution and insertion also use a uniform distribution over the alphabet. The number of editing operations is chosen according to a Poisson distribution (with a mean of 3); the three editing operations have an equal change of being selected. The randomly edited string is included in the negative sample provided it is indeed rejected by the target machine. Otherwise, it is simply discarded.

The random walk algorithm and perturbation procedure serve as building blocks for the generation of training and test samples for each problem. A set of 20.000 strings is first generated as described above; it contains a roughly equal number of positive and negative strings and contains duplicates. The distinct strings are then equally partitioned into two sets, S_0 and S_1, taking care of respecting the positive and negative balance in each one.

- A test sample contains 1500 strings randomly drawn from S_0 without replacement. It never contains duplicates in order to avoid favoring repeated strings in the scoring metric.

- The official training sets are sampled from S_1 with different levels of sparsity (100%, 50%, 25%, 12.5%). They usually contain duplicates, as a consequence of random walk generation from the target machine.

- Training and test sets do never intersect.

6.2.4 Submission and Scoring

Solutions to Stamina problems must be submitted as binary strings to the competition server. The learner is expected to produce a binary sequence of labels where, for each test string, a "1" is added to the sequence if the string is considered to be accepted, a "0" otherwise.

To assess solution accuracy, the sequence is compared with a reference string representing the correct labeling of the test set by the target model. The overlap between the two binary strings is measured through the *balanced classification rate (BCR)*. The harmonic BCR measure was chosen because it places equal emphasis on the accuracy of the inferred model in terms of acceptance of positive sequences and rejection of negative ones. Moreover, it does not require the test set to be balanced in terms of number of positive and negative sequences.

Harmonic BCR is the harmonic mean of two factors. *Sensitivity* is the proportion of positive matches that are predicted to be positive and *Specificity* is the proportion of true negatives that are predicted to be negative. Usually, BCR is computed as arithmetic mean of sensitivity and specificity; the harmonic mean is preferred here because it favors balance between the two factors. Let TP, FP, TN and FN denote the sets of true positives, false positives, true negatives and false negatives, respectively. Sensitivity and specificity are defined as follows:

$$Sensitivity = \frac{|TP|}{|TP \cup FN|}$$

$$Specificity = \frac{|TN|}{|TN \cup FP|}$$

$$BCR = \frac{2 * Sensitivity * Specificity}{Sensitivity + Specificity}$$

A problem is considered broken if the BCR score obtained is greater than or equal to 0.99. As in Abbadingo, hill-climbing is made almost impossible by a binary feedback from the oracle, depending on whether the problem is broken or not. A cell is considered broken if all its five problems are broken. The Stamina website dedicates a private section to each registered participant that provides visual feedback about the performance of her algorithm(s). In this section, problems and cells of the submission grid turn to green when broken.

	Sparsity			
$\|\Sigma\|$	**100%**	**50%**	**25%**	**12.5%**
2	0.99 (1)	0.95 (1)	0.67 (3)	0.66 (3)
5	0.97 (1)	0.78 (2)	0.59 (4)	0.52 (4)
10	0.93 (1)	0.64 (3)	0.51 (4)	0.50 (4)
20	0.91 (1)	0.63 (3)	0.54 (4)	0.51 (4)
50	0.81 (2)	0.64 (3)	0.57 (4)	0.50 (4)

Table 6.2: Average BCR of Blue-fringe in each cell; the difficulty level is shown in parenthesis.

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

Table 6.3: Calibrating cell difficulties, based on the scores given in Table 6.2

6.2.5 Blue-fringe baseline

Blue-fringe has been ran on all Stamina problems in order to adjust the grid for feasibility and fix the difficulty level of each cell.

Adjusting the grid is similar in spirit to what has been done for Abbadingo. The idea is to adjust free parameters of the competition design in such a way that a state-of-the-art algorithm, here Blue-fringe, breaks the easiest problems. Adjusted parameters were the sizes of the training and test sets and the average length of the strings with respect to the automaton depth. This trial-and-error process converged with three problems broken in the easiest cell; reasonable scores were also observed for Blue-fringe on the two remaining problems and in adjacent cells.

The performance of the Blue-fringe baseline is summarized in Table 6.2 and illustrated with convergence curves in Fig. 6.1. As shown there, the BCR score decreases along each of the two difficulty dimensions; this experimentally confirms the expected effect of an increasing alphabet size on the induction difficulty.

Thanks to those scores, the difficulty level of each cell has been calibrated using the rules defined in Table 6.3. The notion of cell difficulty proved useful for driving the competition. Indeed, the winner was the first technique to have broken a hardest cell. It is not used anymore since the competition has evolved into an evaluation platform with exact scoring (see Section 6.4).

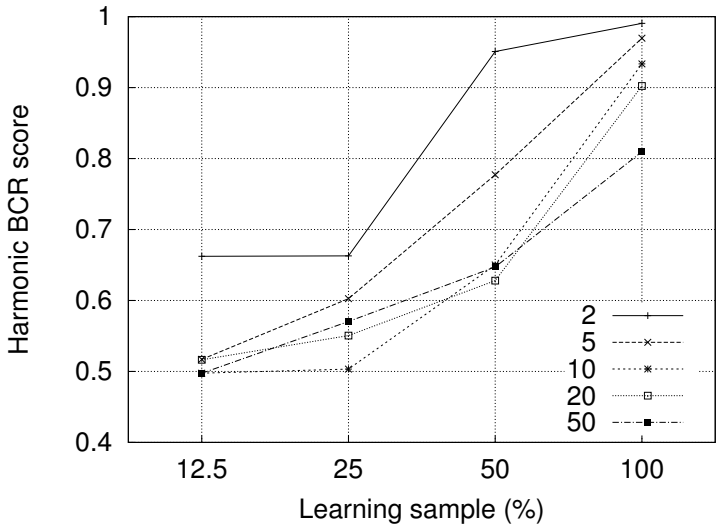


Figure 6.1: Performance curves of Blue-fringe.

6.3 Competition results

The Stamina competition started on March 2010 with December 31 as official deadline for submitting results. Between these two dates, 1856 submissions were made by 11 challengers. Among them, 61 have a BCR score of at least 99%, breaking 42 different problems in 10 different cells.

The competition hall of fame is shown in Fig.6.4. A quick temporal review shows the following:

- The easiest cell (alphabet 2, sparsity 100%) has been broken only five days after the competition start. This first result was achieved by Manuel Vázquez de Parga Andrade with the Equipo algorithm; the latter learns automata teams [Gar10]).

- This first result has been followed by an apparent quiet period in the competition. Some challengers were actually submitting, without successfully breaking new problems and cells. Therefore, in the absence of a more detailed hall of fame page, no activity was visible on the competition website.

- A few weeks after, Marijn Heule and Sicco Verwer, with the DFASAT algorithm (described later), started solving all cells of difficulty 1 in the left-most column.

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

Table 6.4: Stamina hall of fame. Broken cells are annotated with the winner name. In other cells, the number in italics indicates how many of the five problems were broken on December 31, if any. Difficulty levels in parenthesis are recalled from Table 6.2.

- After a new apparent lull, they eventually broke the first cell of difficulty 2 (alphabet 50, sparsity 100%), thereby taking the head of the competition.
- They eventually won the competition with an extra cell broken (alphabet 50, sparsity 50%). This was the only cell of difficulty 3 broken during the competition.
- Individual problems were also broken in cells of the second column (sparsity 50%), as shown by numbers in Table 6.4.

As the same table shows, no problem has been broken in the last two columns, that is with a sample sparsity of 25% or less. This fact must be interpreted with caution.

- While fewer activity has been monitored on such cells, a few approaches actually performed quite well, in particular above the baseline.
- Because of specific strategies used by the different participants, only partial results were available at the end of the competition. In other words, participants have not submitted results on all available problems, making it difficult to compare and draw conclusions.

6.3.1 The winning algorithm

DFASAT reuses the baseline algorithm in an appropriate way but relies on additional key ingredients, namely,

- An adapted heuristics for scoring candidate merging operations;

- A random perturbation of such heuristics;
- An intense search after original reduction to a satisfiability problem (hence the DFA**SAT** name).

Each of them is described in turn in the following sections.

Regular induction seen as a SAT problem

The idea behind the satisfiability part of DFASAT is to (1) reduce the problem of finding the minimal DFA consistent with a labeled sample to a graph coloring problem; (2) encode the latter into a satisfiability problem; and (c) solve it using an efficient SAT solver.

The principle of reducing DFA induction to graph coloring dates back to 1997, more or less like the Abbadingo contest itself – see, e.g., [Cos97]. It consists of coloring PTA states with a minimal number of colors such that two states that may not merge have different colors. As explained in Chapter 4, two PTA states may not merge if their continuations are incompatible.

The main drawback of this approach is the number of constraints which may become huge for big samples. This can lead to coloring problem instances that are impractical in practice. In spite of compact Boolean encoding of the graph coloring problem and major improvements in SAT solving in recent years, some of the Abbadingo and Stamina problems remain intractable with this technique only. See [Heu10] for details about graph coloring and SAT translations in DFASAT.

Mixing state-merging and SAT solving

To overcome this tractability problem, the authors reuse classical state-merging upstream their SAT solving technique. The idea here is to execute the first steps of Blue-fringe to first reduce the PTA to a partially identified DFA.

Executing even the first steps of a state-merging algorithm may drastically reduce the number of states of the graph coloring problem, and hence the number of constraints. However, the state-merging step implies that the solution provided by the SAT solver is no longer exact. As the first few merges performed by Blue-fringe are supported by a lot of evidence, they are then expected to lead to an optimal solution in practice provided some good scoring heuristics is used [Heu10].

According to the winners themselves, the Blue-fringe + SAT technique was not sufficient to break Stamina cells above the first difficulty level. As

another technique already had gained a cell of same difficulty, further optimizations were required to win the competition.

Additional adaptations

In order to obtain convincing results on harder cells, a few additional ingredients were added to the procedure:

- DFASAT uses a different scoring heuristics than the one used by Bluefringe. Since Abbadingo it is commonly admitted that the heuristics must be based on the number of merged states sharing the same positive or negative label. To win the Stamina competition, DFASAT used a different one, related to the number of merged transitions sharing the same symbol.

  The explanation relies on the kind of automata considered, especially on large alphabets: as two different states of the target DFA are unlikely to have many outgoing transitions in common, two PTA states having more than a few of them are very likely to correspond to the same target state.

- Instead of considering only one solution, DFASAT includes a search strategy over multiple candidates. This seems natural with SAT-based graph coloring since all possible solutions are captured in a compact form. The authors have also intentionally introduced search in two other places.

  - The first involves a random perturbation of the scoring heuristics to reduce the criticality of wrong initial merges potentially made during the first phase.
  - The second is to look for non-minimal consistent automata by considering more colors than actually required for coloring the graph.

    This appears to contradict Occam's razor principle but makes sense in the context of Stamina. The actual problem to solve in the competition is not strictly speaking the identification of target automata but the learning of good approximate solutions reaching 99% of BCR score. Looking for automata of about 50 states is probably a better criteria than looking for the minimal consistent one.

6.4 Evolution towards an evaluation platform

In the spirit of Abbadingo, the competition website is still available online. It is aimed at becoming an online benchmark for evaluating novel induction techniques. To achieve this goal a few changes were made on the competition server:

- The average score obtained by Blue-fringe on each cell has been published in the documentation section of the website. Moreover, the cell difficulty level has been made obsolete; it is only kept for documentation of the competition itself.
- The public hall-of-fame has been updated. Instead of displaying the winners of broken cells only, each cell is annotated with the three best challengers in descending order of their average score for that cell. Their score is also disclosed.
- The oracle has been modified to provide the exact score obtained when attempting to break a problem. The private submission grid of each participant displays the best score obtained on each problem for which she submitted.

These choices were made to provide a more transparent feedback while keeping a competitive aspect to the platform. Future work is worth considering:

- Interactive inductive techniques, e.g. QSM, could hardly compete so far due to the absence of an online oracle answering membership queries. Implementing such an oracle presents no particular difficulty; it would however require generating fresh new problems to avoid interfering with the current grid and challengers already competing on it.
- White box benchmarks, where target machines would be disclosed together with samples, might also be envisaged. This would help evaluating inductive techniques that use domain knowledge; the target machines prove useful for simulating such information. Alternatively, sharing binaries for generating new target machines and samples could help reusing the Stamina protocol.

Summary

This chapter discussed *Stamina*, an evaluation platform for inductive model synthesis. Stamina follows a previous platform from the machine learning

community known as Abbadingo. Our work on Stamina sets a basis for overcoming limitations of Abbadingo for evaluating inductive model synthesis techniques such as QSM and ASM. In particular, random automata in Stamina are more representative of the state machines found in software engineering problems.

Stamina was first designed to be a formal competition in automaton induction before evolving into an online benchmark. The competition was won by DFASAT, a novel induction algorithm that significantly pushed forward the state-of-the-art by outperforming Blue-fringe used as baseline. The design of Stamina make the competition results especially relevant for our software engineering standpoint. In particular, DFASAT opens important perspectives for inductive model synthesis in our application domain.

Chapter 7

Tool Support

This chapter discusses tool support for the techniques presented in the previous chapters. Section 7.1 highlights the key points of our model checker for guarded hMSC and guarded LTS, thereby pursuing the discussion from Chapter 3. Section 7.2 presents the ISIS tool, an integrated inductive model-synthesis approach within the "scenarios, goals & state-machines" triangle. This tool implements techniques from Chapter 4. Section 7.3 presents another toolset aimed at modeling and analyzing process models captured with guarded hMSCs. This toolset provides support for the analyses detailed in [Dam11].

7.1 A model checker for process models

Our model-checker for guarded hMSC extends the model-checking technique presented in [Gia03]. The latter supports model-checking of LTS state machines against FLTL safety properties. Our tools extends this to the model-checking of g-hMSC and g-LTS models.

The model-checking technique is outlined on Fig. 7.1. Let us look at it backwards from right to left:

- Roughly, compositional model-checking consists in searching for an accepting state in the composition of two automata: one for the model being checked (top), the other for the negation of the checked property (tester automaton at bottom). Our tool is limited to safety properties, for which such tester always exists (see Section 2.6.3).
- The model automaton is synthesized from the g-hMSC and fluent definitions using the techniques described in Chapter 3. At first glance, it is a pure LTS capturing the trace semantics of the input g-hMSC (see below).

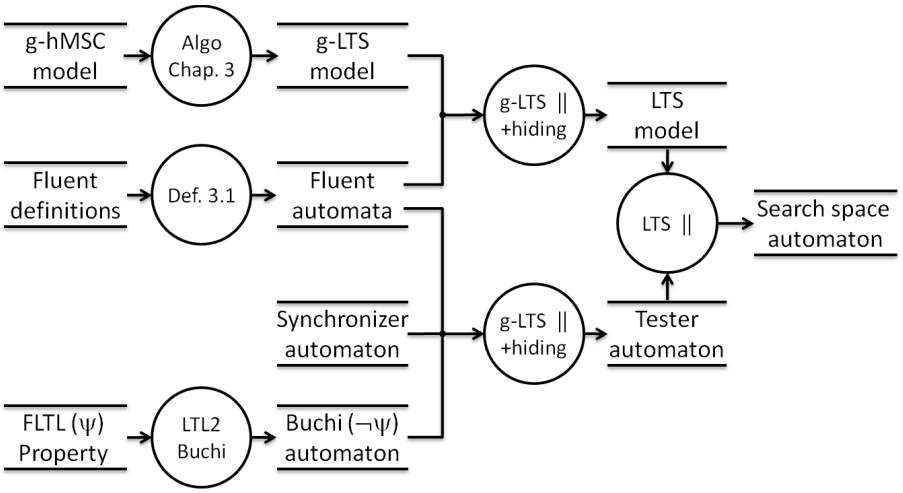


Figure 7.1: Model-checking guarded hMSC: automata compositions

- The tester automaton is synthesized using the technique described in [Gia03]. Roughly, it consists in translating the negation of the checked safety property into a Büchi automaton [Gia02]; the latter is then composed with fluent automata and with a synchronizer automaton forcing the transition on the Büchi automaton after every system event.

The main differences between our model-checker and the technique described in [Gia03] and its LTSA implementation are the following.

- Unlike LTSA, our tool makes use of *guarded* automata instead of simple LTS. Guarded automata are a flavor of guarded LTS that distinguishes between accepting and non-accepting states.

  Guarded automata allow us to capture standard automata, Büchi automata, fluent automata, LTS and g-LTS through one single data structure. A single composition algorithm is sufficient for apparently different composition operators in Fig. 7.1.

  The composition operator on guarded automata extends the one described in Section 3.2.2 by making a distinction between accepting and non-accepting states. This operator reduces to g-LTS composition when the composed automata have accepting states only; it reduces to LTS composition if they don't have guards.

  The composition operator requires computing guard conjunctions and checking their satisfiability. Our tool uses Binary Decision Diagrams (BDD) to implement this efficiently [Bry86].

- LTSA checks temporal properties for a specific initial state, specified through an initial value for each fluent. Guarded models use an ini-

tial condition C_0 instead (see Sections 2.7 and 3.2). Such condition actually captures a class of fluent initial states.

Therefore, our tool must check the temporal property for any initial state satisfying the initial condition. This requires the definition of fluent automata given in Section 3.3.2 instead of the one given in [Gia03]; the synchronizer automaton must also be slightly adapted so as to first synchronize with guarded transitions from initial states of the fluent automata.

- For effective feedback in case of a property violation, our tool needs to keep track of the initial fluent assignment during search. To achieve this, the model and tester automata are not pure LTS; instead, they have fluent value assignments on all transitions from their initial state. These transitions capture the choice of a fluent initial state at the beginning of every trace (see Section 3.3.2).

Discussion

Our tool is implemented in Java 1.5.0. It has been tested on a few case studies (see, e.g., [Dam10] and [Dam11]).

Numerous improvements could be made:

- Our model-checking procedure relies on three costly compositions: one for building the model, one for the tester and one for the model-checking itself (see Fig. 7.1).

  The third composition is implemented as a search: it ends and returns a trace as soon as a property violation is found instead of capturing the composed automaton in a data structure and analyzing it afterwards. In contrast, our tool explicitly capture the tester and the model automata.

  There are pros and cons in doing so instead of a whole search strategy covering the three compositions:

  - On the negative side, verifying a property requires explicitly capturing the whole state space of the guarded hMSC; this is known to be exponential in the number of fluents (see Section 3.3.2).
  - On the positive side, our strategy proves more efficient when multiple safety properties are checked on the same process model. Indeed, the g-hMSC may be synthesized as a LTS automata only once; moreover, the latter can be minimized. A search strategy would not benefit from this state space reduction.

- Unlike LTSA, our tool does not support model-checking of liveness properties; it does not implement partial order reduction optimization either. The procedures described in [Gia03] could certainly be adapted to guarded models.

7.2 The ISIS Synthesizer

The interactive LTS synthesis technique described in Chapter 4 has been implemented in the ISIS tool (Interactive State machine Induction from Scenarios). The tool integrates the following features.

- Editing and visualizing scenarios in a graphical syntax à la MSC (see Section 2.4).
- Definition of fluents and goals, on a per-agent basis (see Sections 2.5 and 2.6).
- Verification of a multi-view modeling specification, more specifically consisting in checking the consistency of scenarios, state machines and goals (see Sections 2.4.5 and 2.6.2).
- Interactive induction of LTS state machines from scenarios, taking fluents and goals into account (see Chapter 4).
- Decoration of scenario timelines and state machines with state invariants (see Section 2.5).
- Induction of safety properties from scenarios (see [Dam11]).

Figure 7.2 shows a snapshot of the ISIS tool. On the left, the current scenario specification is outlined: the agents forming the system are listed together with associated fluent and goal definitions; positive and negative MSC scenarios are listed below. On the right, two scenarios are shown for the train system case study. The scenario "Normal journey" in the background is annotated with computed state invariants.

The analysis and synthesis techniques in ISIS are made available "on demand" by right-clicking the name of a specification (see Fig. 7.3).

Check specification allows the user to check if the collection of positive and negative scenarios is consistent (as defined in Section 2.4.5) and that the scenarios do not violate known goals (as defined in Section 2.6.2).

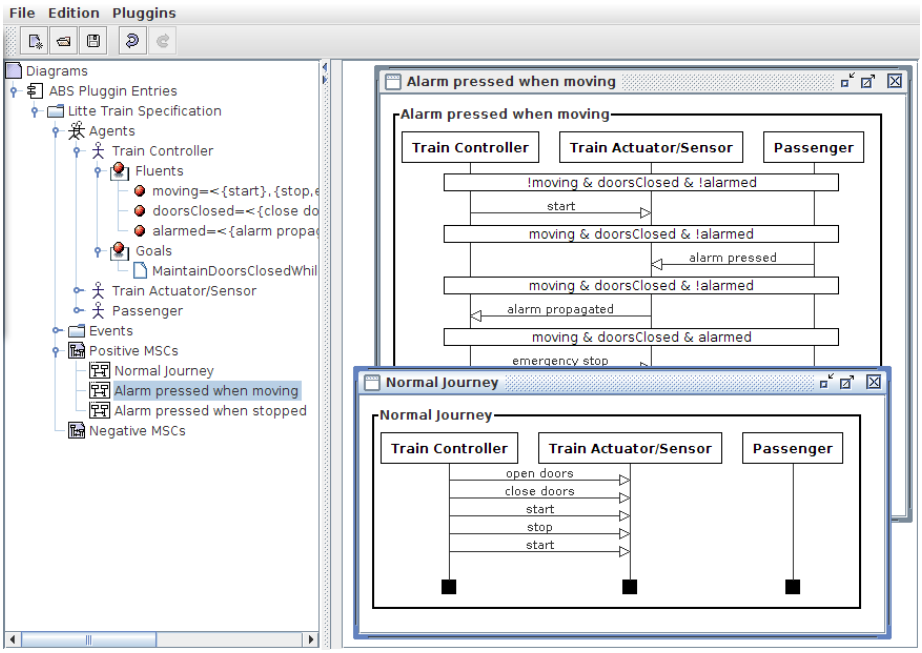


Figure 7.2: The ISIS tool, scenarios of the train system case-study

Synthesize LTSs launches the interactive induction process described in Chapter 4. ISIS implements the QSM algorithm; the scenario specification is automatically completed with the user answers to generated scenario questions (see Section 4.3). Moreover, the induction is automatically constrained with available fluents and goals (see Section 4.4). The tool does not support hMSC models; therefore it does not integrate ASM either (see Section 4.5).

Compose System LTS builds the LTS state machine of the composed system and opens it in the right pane.

Discover goals launches the techniques allowing to discover of goals from scenarios as discussed in [Dam06, Dam11].

During inductive synthesis, scenario questions are submitted to the end-user for classification (see Fig. 7.4). When rejecting a scenario, a "No, why?" button invites her to provide the rational for rejection by updating fluent and goal definitions.

At the end of the synthesis process, new scenarios appear in the collection according to their classification by the end-user. Those scenarios are given comprehensive names, such as "Opening doors while moving" (see left pane in Fig. 7.5).

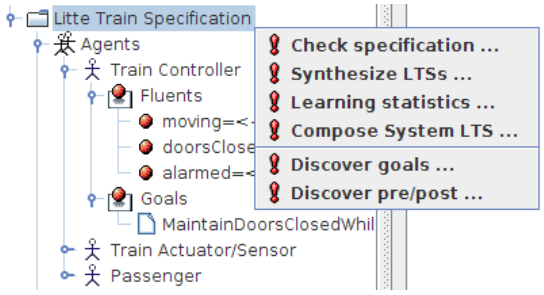


Figure 7.3: Available analysis and synthesis techniques in the ISIS tool

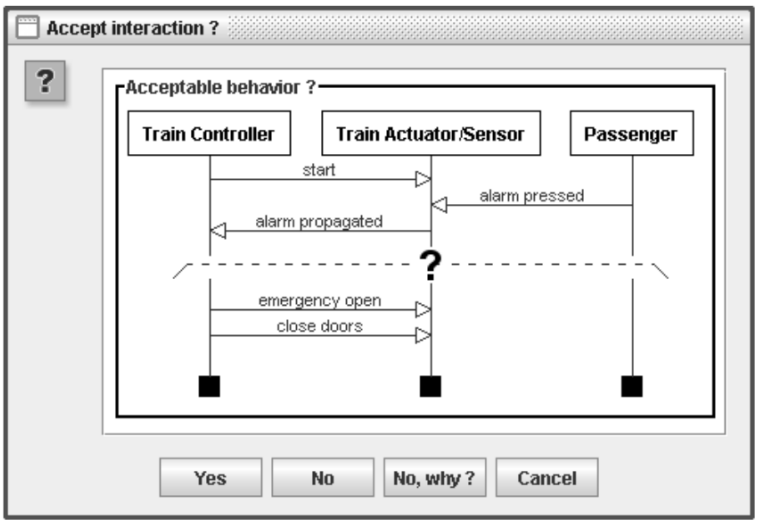


Figure 7.4: A scenario question submitted to the end-user during inductive LTS synthesis

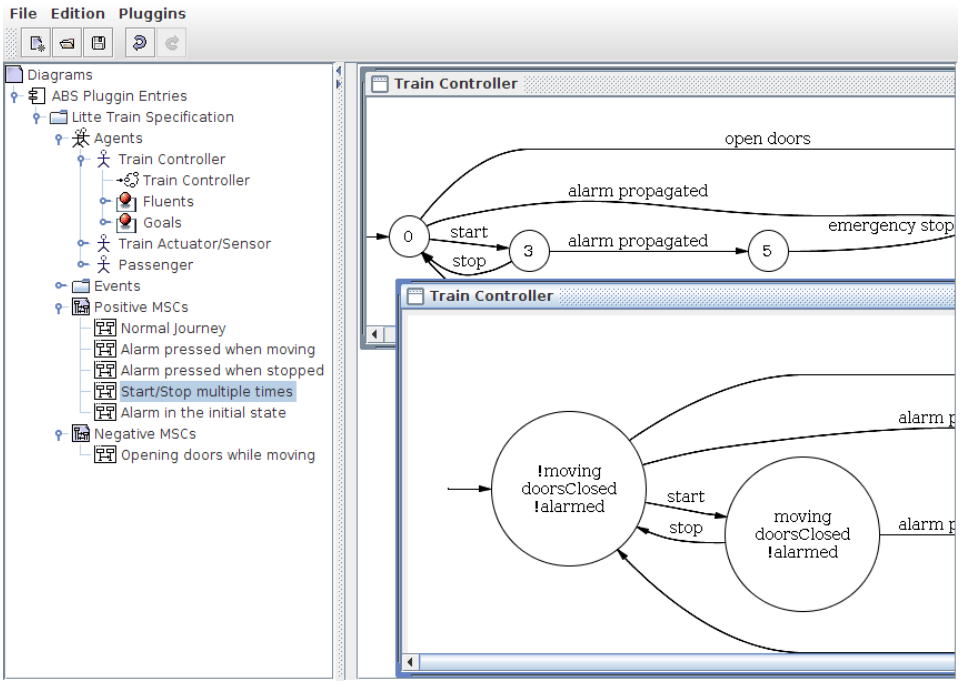


Figure 7.5: After synthesis, the annotated state machines can be visualized.

The LTS state machine for each agent also appears in the specification outline. State machines can be visualized with or without state assertions. The LTS state machine of the train controller is shown in both modes in Fig. 7.5.

Discussion

The ISIS tool has been used successfully on various case studies during the evaluation of the thesis (see Chapter 5). Our integrated approach to multi-view model synthesis proves very effective in practice. The availability of *multiple* synthesis techniques supports the incremental intertwined construction of each view by launching the adequate synthesis assistant.

Our approach could be better supported in practice by complementing the "on demand" approach with additional guidance. The idea here would be to run the different synthesis techniques in background. A list of suggestions could be maintained such as:

- "*new scenarios available for classification, run LTS synthesis!*",
- "*new goals have been inferred, click here*".

Consistency checking of the specification would certainly benefit from a similar feature. The tool described in the next section makes an investigation into this kind of tool support.

7.3 The Gisele process model analysis toolset

Safety-critical medical therapies call for well-defined processes. In particular, a clinical pathway is a documented process, based on medical protocols, guidelines and recommendations, centered on a specific patient class with similar needs. It generally involves multi-disciplinary teams and addresses clear clinical goals [Mid00].

The toolset presented in this section was motivated by the need for automated support to the building and analysis of models of clinical pathways and, more generally, mission-critical process models involving decisions.

Guarded hMSCs were defined as a suitable process modeling language accessible to stakeholders; it has a formal semantics enabling a variety of model analysis [Dam09, Dam11].

Those analyses are made at the g-LTS level; the toolset here thus relies on our synthesis algorithm for deriving g-LTS from g-hMSC, described in Section 3.3.1.

A snapshot of the main screen of toolset front-end is shown in Fig. 7.6. A description of the toolset in action on a real medical case study can be found in [Dam11]. This section discusses a few design decisions and highlights key points of the implementation. The main facilities provided by the toolset are the following:

- Editing and visualization of process models made of tasks and decision nodes defined on fluents and tracking variables (see Section 2.7 and [Dam09, Dam11]). Tasks can be successively refined in sub-processes, as illustrated in Fig. 7.6.
- A variety of analyses, including (see [Dam11] for details),
  - Checking the completeness, non overlapping and satisfiability of guards at decision nodes.
  - Checking the adequacy of decision nodes, that is, whether these are made on fresh, accurate variable values.
  - Verifying pre-conditions of tasks.
  - Verifying non-functional process requirements such as time constraints, dose constraints, cost constraints, etc.

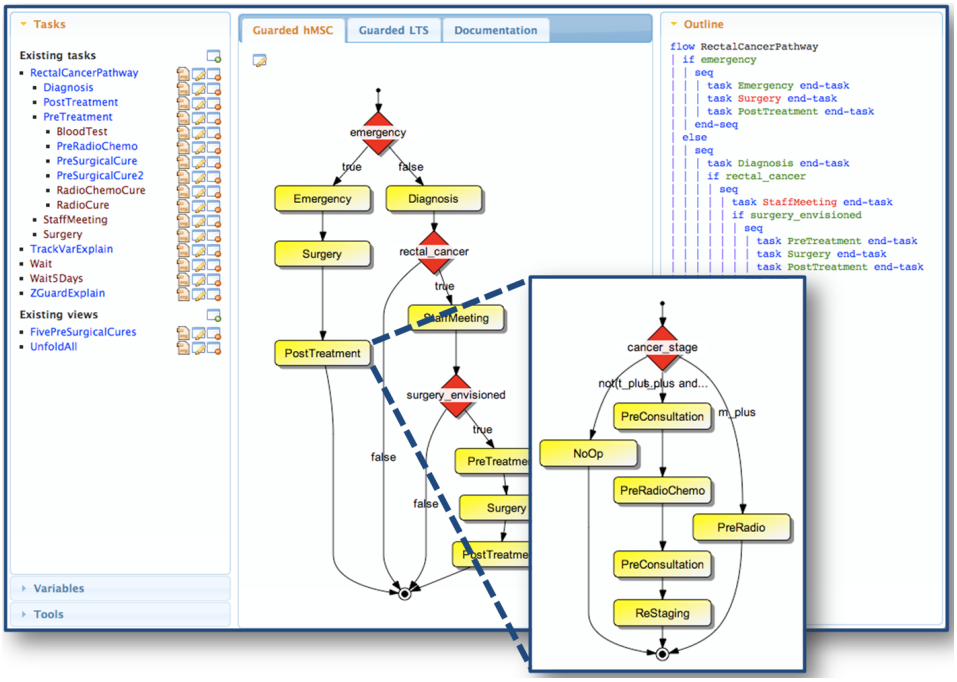


Figure 7.6: The Gisele tool, a clinical pathway analyzer

- Visual means for eliciting process models; documenting them with stakeholders; unfolding process models for specific analyses; projecting them on specific patient classes, and so on.

Design decisions

The toolset has been designed for (1) eliciting critical processes during stakeholder interviews (2) analyzing them in real-time and (3) providing browsable process documentation. This implied two main design decisions:

Textual input For allowing a rapid capture of process models by an analyst, a simple textual language is used as input language instead of a graphical language. This language essentially amounts to a guarded command language with constructs for iteration, loops, deterministic and non-deteministic choices, etc.

For example, the meeting scheduling process can be translated to this textual language as shown in Fig. 7.7.

```
workflow MettingScheduling do
  InitiateMeeting
  AcquireConstraints
  until not(date_conflict) or second_cycle
    Arbitrate
    if resolve_by_weakening
      WeakenConstraints
    else
      ExtendDateRange
      AcquireConstraints
    end
  end
  ScheduleMeeting
end
```

Figure 7.7: Meeting scheduling process encoded in the Gisele tool

Real-time proactive mode Unlike the ISIS tool introduced in Section 7.2, a proactive mode is used instead of reactive one with respect to checks and feedback:

- Reactive mode: in the ISIS tool, synthesis and checks are made "on demand": for example, the user explicitly launches consistency checks and receives dedicated feedback as a result (see Section 7.2).
- Proactive mode: in the Gisele tool, the end-user navigates through the model thanks to the GUI that runs inside a web browser. Every time she looks at a task, the tree on the right panel provides real-time feedback about checks and their results, "green" means success, "red" means fail. The idea is inspired from continuous compilation chains in integrated development environments (IDE) such as Eclipse [Gam03]. Our experience suggests that this provides a natural and effective guidance in terms of continuous improvement of a process model.

Architecture

The architecture of the toolset is shown in Fig. 7.8. The main modules are briefly described below:

Web GUI This module is the graphical user interface (GUI), that runs mostly inside a web browser. It is implemented in HTML5, CSS3 and Javascript [Pil10] and uses Scalable Vector Graphics for displaying guarded hMSCs [Eis02]. Thanks to the web services offered by the "proactive process model" module, the implementation of the GUI amounts to a simple model navigator, implemented internally with an MVC-like design pattern.

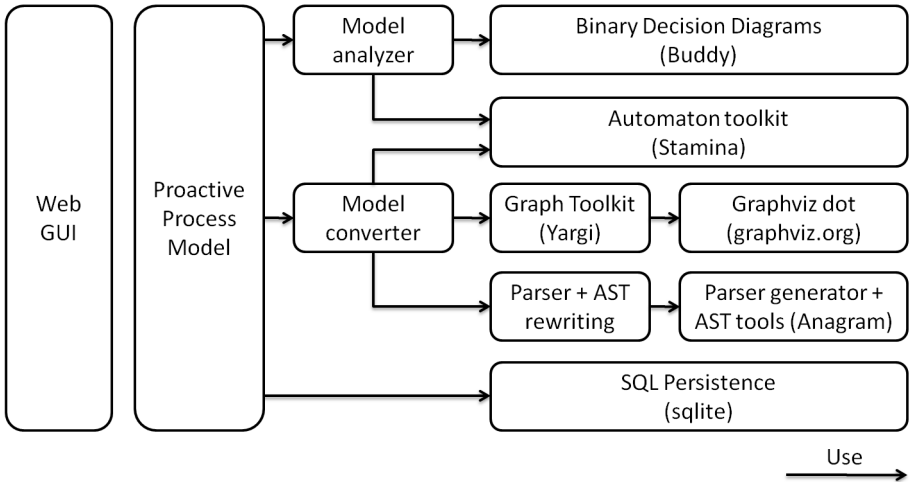


Figure 7.8: Architecture of the Gisele tool

The main screen of this navigator is made of three panels (see Fig 7.6).

- The left panel gives a global outline of the process model and allows navigating it through. Links are provided to open and update task and fluent definitions.
- The current task is displayed as a process graph in the middle panel; it is implemented in SVG. The layout of the graph is automatically computed server-side with the well-known *dot* utility[1]. Graph nodes can be clicked to navigate through task refinements.
- The current task is also outlined through an annotated syntax tree in the right panel. Annotations provide analysis real-time feedback to the user; tasks and decision nodes for which at least one check fails are displayed in red. Clicking on a node in this tree gives a detailed analysis explanation and counterexamples showing why check fails.

Proactive process model Process models are made of several tasks in successive refinements, textual definitions such as the one in Fig. 7.7 for each of them, fluent definitions, task documentation, etc. This module implements the structure of such models which are kept persistent through the use of the SQLite database engine; the latter also provides support for model queries through SQL [Dat97].

In addition, this module implements the proactive mode described in the design decisions through a design pattern inspired from logical data independence [Dat03]. Roughly, this pattern consists in hiding the difference between "base" information and "computed" one.

[1] available at http://graphviz.org

For example, the precondition is a base attribute of a task whose value is chosen by the analyst; the fact that this precondition is met or violated is another Boolean attribute whose value is computed as the result of a dedicated precondition check [Dam11].

Systematically hiding this distinction between base and computed information helps implementing the proactive mode. Indeed, user feedback can be implemented through model queries. For example, "what are all tasks whose precondition is violated?". The module automatically triggers the model checks if needed to answer such query. To achieve this, it relies on other modules, which are described below.

Model analyzer This module implements the analyses detailed in [Dam11]. They rely on various instantiations of an abstract fixpoint algorithm on g-LTS. Such analyses makes an intensive use of automata and binary decision diagrams with the help of dedicated libraries. The automaton toolkit is the one implemented for the Stamina contest (see Chapter 6). The *Buddy* library is available at `http://buddy.sourceforge.net`; being implemented in C, we use a interface binding for ruby available at `http://people.cs.aau.dk/~adavid/BDD/`[2].

For every process and sub-process instance, the proactive module automatically triggers the execution of an analysis as soon as its results are needed for answering a model query. Those results are kept as decorations on g-LTS states. The decorated states are kept as a cache for analysis results. This cache is cleaned up when a process definition is changed, by simply throwing its g-LTS away.

Model converter The implementation of the proactive module relies on specific algorithms for rewriting processes under multiple forms. For example, the textual definition of a task is parsed and kept as an abstract syntax tree (AST); rewritten as a guarded hMSC; converted to a graph for being displayed; compiled into an equivalent guarded LTS for analyses; and so on.

This module implements conversion algorithms, relying on external libraries for parsing, manipulating ASTs, graphs and automata. This module also provides traceability support between the multiple views of the same process instance.

[2]both have been last retrieved on September 13 2011

Chapter 8

Related Work

This chapter reviews relevant work on behavior model synthesis and related techniques for model analysis and goal inference. We also briefly discuss a few goal inference techniques that nicely fit our formal framework. More specifically, Section 8.1 reviews approaches aimed at synthesizing state machine models inductively from scenarios. Section 8.2 discusses related work on deriving behavior models for analysis of scenarios and process models. Section 8.3 then discusses a few approaches that use inductive synthesis as a means for inferring requirements and exploring system designs.

Model synthesis is a fairly broad topic. In particular, this chapter will not discuss approaches falling outside the scope of the thesis such as controller synthesis [Cla82, Pnu89, Asa95], program synthesis [Man71, Bal85, Was03], model-based code generation [Koh00, Was03], and model-based reverse engineering [Bri03, Yu,05].

8.1 Inductive synthesis of behavior models from scenarios

This section compares our inductive approach introduced in Chapter 4 with existing techniques for synthesizing state machines from scenarios. A more complete survey of those techniques can be found in [Lia06].

8.1.1 Statecharts synthesis from sequence diagrams

Whittle and Schumann proposed a technique for generating UML statecharts from sequence diagrams that capture positive scenarios (and positive scenarios only) [Whi00]. Their technique requires scenario interactions to be explicitly annotated with pre- and post-conditions 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 [Kru00]. Their algorithm also requires state information, through MSC conditions, as additional input.

Those techniques generalize behaviors described in input scenarios by introducing sequencing, alternatives and loops. State annotations provide the semantic basis for guiding the generalization process. This contrasts with a grammar induction approach since, in that case, behavior generalization is driven by compatible state merging, which is semantically rooted in *event-based* continuations of PTA states.

In comparison with those techniques, our approach has the following strengths:

- State machine synthesis techniques from scenarios are often in the requirements engineering phase, where the target system is unknown [Wei98]. Event-based scenarios are likely to be more easily provided by end-users than operation specifications in terms of pre- and post-conditions, MSC conditions, etc.

  Our technique uses a very simple input scenario language. In contrast, the techniques in [Kru00] and [Whi00] require state annotations; the user of the technique is expected to provide these. In our case, fluent annotations are used to prune the induction process when available; however, they remain optional.

- As discussed in Chapter 4, illustrated in our evaluations (Chapter 5), and further discussed in our toolset presentation (Chapter 7), our algorithms support a flexible synthesis approach. This approach already works with limited input and adapts to richer ones as they are typically made available in subsequent analysis phases and design iterations.

  In particular, our approach supports simple MSC scenarios as input while also working with structured forms of scenarios such as hMSCs. Similarly, fluents and goals are not required in the first place but are intended to smoothly integrate state-based knowledge in an incremental way.

8.1.2 Minimally adequate teacher approach

Makinen and Systa developed an interactive algorithm for synthesizing UML statecharts from sequence diagrams [Mak01]. This is another approach relying on grammatical inference techniques for generalizing examples of system behavior captured in scenarios. This work is inspired from the approach of minimally adequate teacher and the L^* algorithm proposed by Angluin [Ang87].

As with our approach, and unlike those discussed in the previous section, the technique works with scenarios only and does not require additional state-based annotations. There are significant differences with our approach though.

- Their input sequence diagrams capture positive examples of system behavior only. The negative knowledge required to avoid overgeneralization comes from negative answers to membership and equivalence queries asked to an oracle (see below); such negative knowledge does not take the high-level form of end-user negative scenarios.
- Grammar induction is applied on a per agent basis. In other words, induction traces are sequences of events seen by a single agent along its scenario timeline. The alphabet of possible events and the learning strings are specific to each agent; the learning problem is tackled separately for each of them.
- The interacting user is asked to answer both membership and equivalence queries. Membership queries are traces to be classified as positive or negative examples of agent behavior; they amount to our scenario queries (in a less user-friendly form). Equivalence queries, in contrast, require the ability to classify state machine candidates as correctly capturing the complete behavior of a specific agent.

As discussed in Section 4.7, tackling the induction problem on a per agent basis is an interesting alternative to our approach. For end-user involvement, however, a per-agent approach seems less appropriate than ours; it requires the user to be able to classify traces in a different target language for each agent. It appears much more convenient for the end-user to interact in the input scenario language of the global system as this is the language she used in the first place.

Equivalence queries may also prove difficult to answer by an end-user in practice as the agent state machines are unknown. That being said, a certain form of equivalence query might be seen in our approach as well. With the ISIS tool, for example, a visual inspection of agent and system state machines is recommended for validation and the triggering of a new modeling cycle if required. Such inspection may typically highlight negative scenarios in case of overgeneralization or new positive scenarios illustrating additional desired features (see Section 5.2.2).

As their approach is interactive and can be used with only a few scenarios, it proves useful in the early phases of system design. Unlike ours, though, the approach does not adapt to richer scenario inputs. It does not take additional knowledge into account such as fluents or goals. In other words, multi-view consistency is not guaranteed by their synthesis process.

8.1.3 Play in/Play out with Live Sequence Charts

Harel et al. proposed a technique in [Har05] for synthesizing statecharts from Live Sequence Charts (LSCs) [Dam01]. LSCs are an extension of MSCs that distinguish between allowed and required behaviors. In other words, LSCs allow both *existential* scenarios (behaviors that *may* be exhibited by the target system) and *universal* ones (behaviors that *must* be exhibited) to be captured.

LSCs, and the underlying statechart synthesis technique, are used in the Play in/Play out approach for iterative system analysis and design [Har03b, Har03a]. In this approach, LSCs model all desired system behaviors, providing a complete design for the system. The approach may be summarized as follows:

- A LSC specification is built or completed in a *Play in* phase. The user interacts through a graphical user interface (GUI) of the target system, without seeing the LSCs themselves. That is, a *Play engine* allows the user to build and manipulate GUI components. The engine automatically builds LSCs capturing stimuli played in the GUI together with expected responses from the system as specified by the user. Additional tools allow introducing loops, conditions, and other high-level constructs in played-in scenarios.

- In a *Play out* phase, the LSC specification is executed. In contrast to the *Play in* phase, the specification is executed here as if it was the implemented target system. This allows identifying design errors early, behaviors incompletely specified, etc.

- Scenarios played out are captured through LSCs. The latter therefore yield new examples and counterexamples for trigerring a new *Play in* phase. The approach thus provides an iterative and interactive way of building reactive systems through scenarios.

The Play in/Play out engine and our ISIS tool are somewhat similar in that both provide integrated approaches for iterative system design through scenarios. Important differences however exist between the underlying approaches.

- The Play in/Play out approach constrats with our multi-view modeling approach. LSC scenarios are the only models seen by the user; they are expected to capture the complete system specification. This is rather different than making a clear distinction between scenarios, state machines and goals.

  Approaches capturing different system facets through different models prove useful in contexts where end-users are involved. Indeed, they

lead to modeling languages that tend to be simpler to use while enforcing a separation of concerns.

In particular, Live Sequence Charts have a rather complex execution semantics. This makes them harder to use and validate by end-users than Message Sequence Charts.

8.2 Deriving behavior models for analysis

Section 8.2.1 focusses on the analysis of process models. Section 8.2.2 discusses approaches to the model-checking of scenario specifications. Section 8.2.2 briefly discusses how implied scenarios can be used for validating structural system designs.

8.2.1 Analyzing process models

The technique detailed in Chapter 3 for capturing the trace semantics of g-hMSC models aims at enabling a variety of analyses on process models. The model checking of g-hMSCs was discussed in Section 7.1.

A variety of analysis techniques on such models are detailed in a companion thesis [Dam11]. They include the analysis of guards at decision points, including their completeness, disjointness and satisfiability; task precondition checks; analysis of non-functional requirements on the process about timing, cost or doses. All these techniques are performed at the g-LTS level (see Section 3.2). They use various instantiations of a fully generic g-LTS state decoration algorithm [Dam11] briefly mentioned in Section 2.5.2.

Many process languages were proposed in the literature, e.g., UML Activity Diagrams [Obj04], YAWL [Van05], BPMN [Obj08] and Little-Jil [Cla08]. The main advantage of g-hMSCs over these is their emphasis on *decisions-based* processes, where decision nodes regulate outgoing branches through guards on fluents and variables tracking the state of the environment in which the process operates.

Efforts were recently made to adapt verification technology to process models. Typically, a state machine model is derived from the input model and then checked against properties. For example, structural consistency constraints on UML activity diagrams can be checked using the NuSMV model checker [Esh06]. Similar constraints can be verified on Little-JIL process models after task conversion into LTS [Ler04]; this technique uses the LTSA model checker [MK99]. LTSA was also used for deadlock analysis and model-checking of workflow schemas formalized in FSP/LTS [Kar00].

8.2.2 Model-checking of scenario specifications

Theoretical results for the model-checking of Message Sequence Charts were pioneered by Alur et al. [Alu99, Alu00]. These results provide a basis for practical approaches to model checking through automaton synthesis; the authors, however, do not provide practical algorithms for doing so.

The semantic links between MSC, hMSC and LTS appeared in [Uch01b, Uch01a], enabling the model-checking and animation of hMSC models in LTSA [Uch03, MK99]. In particular, a "synthesis" algorithm yields FSP specifications for capturing the traces of each agent involved in a hMSC (see Section 2.4.4).

This technique might be seen as a way of synthesizing agent state machines from scenarios. However, it is derivational rather than inductive; no behavior generalization is involved. In other words, it amounts to refactoring a set of known behaviors without discovering new ones. Moreover, the state machines are not intended to be seen or manipulated directly by the end-user. Our work borrowed the synthesis algorithm from [Uch03] as definition for the operational semantics of hMSC. This algorithm also provides a practical way of model-checking them.

8.2.3 Deriving implied scenarios for structural validation

Implied scenarios are behaviors that are not explicitly found in scenario specifications but can be shown to appear in system implementations meeting those specifications [Alu00, Uch04]. Implied scenarios are inherent to distributed systems; the latter are often described system-wise but implemented component-wise. Capturing all possible interleavings of agent behaviors in scenario specifications is generally impossible.

A technique is described in [Uch04] for validating a hMSC specification with domain experts. For a hMSC H, implied scenarios are defined as traces in $\mathcal{L}_{arch}(H) \setminus \mathcal{L}_{weak}(H)$ (see Section 2.4.4). Implied scenarios denoting maximal traces are enumerated and submitted to the user for classification as positive or negative. Accepted scenarios enrich the initial scenario specification as new examples of desired system behavior. Rejected scenarios are more problematic as they capture traces that will necessarily be exhibited by any system consistent with the hMSC.

This technique can thus be used for validating the system decomposition in terms of agents and their respective interfaces. The presence of rejected implied scenarios is indeed an indication that the system decomposition should be refactored.

As deeply discussed in Section 4.6, implied scenarios may also appear with our techniques. The algorithm from [Uch04] might be easily adapted

to validate system designs in our context; the set of implied scenarios to consider is defined as follows:

$$\mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) \setminus \mathcal{L}(\text{System}),$$

where *System* denotes the system LTS induced by QSM or ASM.

8.3 Eliciting requirements from scenarios

This section addresses the problem of eliciting requirements from scenarios. Section 8.3.1 reviews approaches for inferring goals from scenarios. Section 8.3.2 then discusses how scenarios and safety properties can be used jointly for incremental requirement elicitation.

8.3.1 Goal inference from scenarios

A technique for synthesizing goals from scenario collections was presented in [Dam06, Dam11]. This technique is integrated in the ISIS tool (see Section 7.2). It consists in decorating MSC timelines with invariants on fluents monitored and controlled by the associated agent. From these invariants, goals are induced according to two kinds of specification patterns, namely *Maintain goals* taking the form $\Box(P \rightarrow Q)$ and *Immediate Achieve goals* taking the form $\Box(P \rightarrow \circ Q)$. The inferred goals are submitted to the user for validation.

This synthesis technique helps enriching a multi-view system description while guaranteeing multi-view consistency. An inferred goal is stated as a generalization of the available examples. The accepted goals enrich the goal model accordingly; the rejected ones call for enriching the scenario collection with a counterexample.

The technique relies on the availability of state invariants on fluents; it might be extended to infer goals from any annotated LTS, and hence, from agent state machines or from a hMSC.

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 [van98]. The user here has to provide state-based annotations of scenario interactions. These are the domain pre- and postconditions on the operations operationalizing goals; they are found in the corresponding operation model. Compared with our fluent-based approach, the technique requires more input to be provided to the synthesizer; it requires pre- and postconditions whereas our fluent definitions encode postconditions only.

[Alr07] uses inductive logic programming for extracting requirements from example scenarios and a partial requirements specification. Scenarios capture positive and negative system behaviors whereas the requirements specification captures an initial but incomplete description of the envisioned system. The specification is completed by learning a set of missing requirements that cover all the desirable scenarios and none of the undesirable ones.

8.3.2 Incremental requirement elicitation from scenarios and safety properties

In a multi-view behavior modeling framework like ours, scenarios capture a "lower-bound" on system behaviors whereas goals capture an "upper bound". Scenarios capture behaviors that the system must exhibit whereas goals prune the space of all acceptable behaviors captured by the consistency conditions in Chapter 4, namely,

$$\mathcal{L}^+(\mathit{Scenarios}) \subseteq \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n)$$
$$\mathcal{L}^-(\mathit{Goals}) \cap \mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n) = \emptyset$$

In [Uch07, Uch09], Uchitel et al. developed a technique for capturing within a single behavior model both the lower bound from the scenarios and the upper bound from goals. Modal Transition Systems (MTS) [Lar88] are used instead of LTS for capturing trace-based system behaviors; MTSs are transition systems that distinguish between required, possible and proscribed behaviors.

MTSs support incremental system design through user interactions aimed at eliciting missing requirements. Roughly, the techniques from [Uch03] and [Gia03] are adapted to capture through MTSs (1) a structured form of MSC scenarios and (2) safety properties expressed in a 3-valued variant of FLTL. The obtained MTSs are merged so as to capture the lower and upper bounds of system behaviors in a single MTS; the latter preserves the semantics of both scenarios and goals.

So-called *maybe* traces are extracted from this MTS and submitted for classification by the user (we call them "maybe queries" in the sequel). Such traces denote system behaviors that do not violate safety properties but were not explored in the scenario specification. Accepted traces denote missing scenarios in the specification whereas rejected ones help identifying missing safety properties.

In terms of our formal framework, *maybe* queries belong to the following set:

$$\big(\Sigma^* \setminus \mathcal{L}^-(\mathit{Goals})\big) \setminus \mathcal{L}^+(\mathit{Scenarios}) \tag{8.1}$$

that is, the set of system behaviors that are not explicitly rejected by goals, excluding those already required by scenarios.

When all goals denote safety properties, this language is regular and can be captured through standard automata. Our toolset could therefore be complemented with this interactive synthesis technique.

Even though they rely on a form of scenario question, the approach from Uchitel et al. is not equivalent to our inductive synthesis approach. In particular, no generalization of scenario behaviors occurs in [Uch07, Uch09].

Their approach actually raises questions about the convergence of the underlying elicitation process. Let us assume that we know the expected target system. Let T denote an hypothetical state machine capturing the system behaviors. Multiple scenarios tend to cover overlapping paths of T. However the coverage of each scenario is intrinsically incomplete in terms of event-continuations of T states. When these scenarios are captured through a trace-preserving transition system S, some of its states are trace-equivalent with respect to the target system, but are not equivalent in S. The latter tends to grow when new scenarios are added since no state merging occurs. If T grows, new questions appear and the process might be diverging. This phenomenon seems independent of the *kind* of transition system used to capture behaviors, e.g., LTS, MTS, or standard automata. Indeed, all these automaton flavors have their equivalence relation defined in terms of state continuations.

To sum up, as we understand it, [Uch09] offers no guarantee for the convergence of $\mathcal{L}^+(Scenarios)$ towards the behaviors of the composed system $\mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n)$. As a consequence, even if the safety properties converge towards an adequate set capturing all counterexamples of desired system behaviors, it seems to us that the language characterized by (8.1) will never get empty.

In contrast, under the same conditions where the target system is assumed to be known, grammar induction provides guarantees of convergence towards the target machine T if the input scenario collection is sufficiently growing [Onc93] (see Chapter 4.2). In particular, the identification-in-the-limit framework guarantees that, when its input is rich enough, RPNI induces the target system[1]. Moreover, the available domain knowledge, goals in particular, enrich their input.

These observations open interesting perspectives for future research. The two approaches seem indeed complementary. In particular, $\mathcal{L}(System)$, the language of the system LTS induced with RPNI, tends to converge towards

[1]We restrict our attention to RPNI instead of QSM or ASM here to avoid mixing two different kinds of scenario questions in the discussion.

$\mathcal{L}(Ag_1 \parallel \ldots \parallel Ag_n)$. In other words, *maybe* queries from the language

$$\big(\Sigma^* \setminus \mathcal{L}^-(\mathit{Goals})\big) \setminus \mathcal{L}(\mathit{System})$$

appear promising as an elicitation technique, à la [Uch07, Uch09], but then with a convergence criterion.

Chapter 9

Conclusion

Models play a significant role for elaborating requirements and exploring designs of software systems. They help abstracting from multiple details in order to focus on key aspects of the target system. Complex systems should be captured through multiple models. Each of these focusses on particular facets of the system along its intentional, structural, operational and behavioral dimensions.

Building high-quality models for software systems is not an easy task. To play a significant role, models should adequately represent the essence of the target system; they should be complete enough to capture all its pertinent facets; they should be precise when taken in isolation; they should also be consistent with each other.

This thesis investigated model synthesis as a promising approach for building high-quality models meeting those requirements. Our model synthesis techniques were articulated along two main dimensions:

Vertical synthesis yields the operational semantics of high-level models by deriving lower-level ones. This kind of synthesis makes high-level models amenable to formal analysis.

Along this dimension, the thesis defined an operational semantics for guarded high-level Message Sequence Charts (g-hMSC), a process modeling language used for capturing critical processes involving decisions [Dam11]. This semantics is defined through guarded labeled transition systems (g-LTS); synthesis algorithms were described to derive g-LTS from g-hMSC, and to derive pure event-based LTS from g-LTS. The intermediate g-LTS language proves convenient for avoiding state explosion and for enabling a variety of analyses at that level [Dam11] The implementation of a model-checker for g-hMSC was also described; it adapts a compositional LTS technique borrowed from [Gia03].

Horizontal synthesis yields missing model fragments in multi-view models. It uses the rules of inter-model consistency to produce or complete model fragments from the other views.

Along this dimension, the thesis showed how grammar induction methods can be adapted to synthesize state machines from scenarios that illustrate examples and counterexamples of desired system behavior. Our technique interacts with an end-user who accepts or rejects additional scenarios generated by the synthesizer. Declarative properties such as goals were seen to be an effective way of constraining the induction process so as to avoid undesired system behaviors. The induction space may be pruned thanks to such knowledge, resulting in a faster process, a reduced number of user interactions and a guaranteed consistency of the synthesized state machines with respect to such knowledge.

The model synthesis techniques proposed in the thesis aim at supporting the incremental building of system models and thereby the exploration of requirements and system designs. This objective is met in different but complementary ways.

- Vertical synthesis enables model analysis which in turn proves very effective for systematic model building.

  The operational semantics of g-hMSCs paves the way for systematic construction through model analysis. The thesis proposed a model-checking procedure for process models. In addition, it enables a variety of analyses detailed in [Dam11].

  The potential benefits of such model building approach is illustrated with the Gisele process model analyzer described in Section 7.3. This tool has been successfully used for modeling real, safety-critical care processes.

- Horizontal synthesis achieves similar objectives but in a constructive way.

  Our inductive synthesis technique provides flexible and effective support for behavior model building. It requires only a few scenarios in the first place but also supports structured forms of scenarios in input. In addition to synthesizing state machines that are consistent with the input scenarios, the technique calls for the identification of state variables and goals. It allows the initial scenario specification to be iteratively completed with new examples and counterexamples of system behavior.

  Our approach is supported by the ISIS tool, described in Section 7.2. Its effectiveness has been assessed through various evaluations, both on case studies and on synthetic datasets (see Chapter 5).

As illustrated in the thesis, our specific analysis and synthesis techniques already prove effective for model building when taken in isolation. However, their strengths appear even more clearly when they are combined in an integrated environment. This has been observed when using the ISIS tool on case studies. In addition to inductively synthesizing state machines from scenarios, the tool may infer goals from scenarios, decorate synthesized state machines with state invariants, and perform various consistency checks. A similar effect has been observed in the Gisele toolset which integrates a variety of analyses on process models – see [Dam11] for a convincing case.

9.1 Summary of technical contributions

The contributions of the thesis are summarized below along the horizontal and vertical dimensions of model synthesis.

Horizontal synthesis

- The thesis proposed an interactive procedure for synthesizing agent state machines from positive, negative and structured forms of scenarios. This procedure adapts grammar induction techniques for generalizing scenario behaviors. It is interactive with an end-user in the loop who classifies additional scenarios as positive or negative examples of system behavior. As a side effect, our technique can be used for scenario elicitation.

- A technique was presented for pruning the induction search space. This technique allows injecting various sources of system knowledge into the synthesis process. It was instantiated to integrate fluent definitions, models of legacy components, and goals and domain properties. The pruning technique guarantees the consistency of the synthesized state machines with all such knowledge. It also significantly speeds up the induction process and reduces the number of user interactions.

- Our inductive synthesis technique is supported by a tool. This tool further integrates various checks for the consistency of scenarios, state machines and goals together with a technique for inferring safety properties from scenarios [Dam11].

- The techniques and tools were evaluated on case studies and synthetic datasets. In the latter case, a novel evaluation protocol for inductive model synthesis has been proposed and ran as a formal competition on the Internet.

Vertical synthesis

- The thesis provides a formal semantics for the g-hMSC process modeling language in terms of g-LTS. g-LTS are a kind of labeled transition systems allowing events or guards on transitions. They have a trace semantics in terms of pure LTS together with dedicated composition and hiding operators.
- Two synthesis algorithms were proposed to derive g-LTS models from g-hMSC models and then pure event-based LTS models from g-LTSs. Those algorithms were shown to make decision-based process models amenable to a variety of analyses.
- The implementation of a model-checker for g-hMSC and g-LTS was also discussed. Moreover, the thesis discussed the architecture and important design decisions in the implementation of another toolset aimed at analyzing mission-critical process models.

9.2 Open issues and perspectives

This section lists some open issues with our techniques and discusses a few perpectives for future work.

Incremental building of multi-view models

For incremental model building to converge towards the desired system model, we need to assume that the latter actually exists, at least theoretically. This is arguable, since the system is unknown. During modeling, different alternatives are explored; some design decisions are rejected whereas others are selected on the way towards a better model.

In practice, this means that system modeling is a highly non linear, trial-and-error process. Model refactoring techniques seem therefore important to complement synthesis and analysis techniques. The availability of all of such techniques in integrated environments would support a more effective exploration and design process.

For example, a modeling bottleneck in the ISIS tool is the lack of support for scenario refactoring. This may prevent some design explorations and may even hurt the modeling process when such refactoring is required – due to the presence of negative implied scenarios for example. If such refactoring were to be formally supported, our synthesis technique would allow the user to effectively rebuild state machines from refactored scenarios. The investigation of the intertwined usage of refactoring, synthesis and analysis techniques is thus an interesting perspective for additional support.

This raises the question of what makes user guidance effective in an environment integrating multiple techniques. As discussed in Chapter 7, the ISIS and Gisele tools differ in the way such guidance is implemented. In the former case, the available techniques and analyses are made contextually available and executed on demand. In the latter case, the analyses run in the background; real-time feedback is provided as the user navigates through the process model. Both approaches were shown to present advantages and drawbacks.

Convergence and scalability of the synthesis process

Our choice of grammar induction for incremental state machine synthesis is partly motivated by the convergence criterion it provides towards adequate models of the target system (see Section 4.7). The soundness of such theoretical convergence has been argued to be required for the synthesis of behavior models from both scenarios and goals (see Section 8.3.2). As discussed in Section 4.7, work remains to be done to root our ASM* algorithm on such a sound theoretical framework.

Even if convergence guarantees are provided, QSM raises an issue about the number of scenarios submitted for classification by the end-user. As seen in the experiments, additional knowledge such as fluent definitions and goal specifications help reducing the number of questions to be rejected. However, the number of accepted scenarios does not similarly reduce. This may lead to a scalability problem for interactive synthesis on very large models.

One way to tackle this problem would be to explore new ways of interacting with the user. The latter might for example ask to terminate the induction process earlier. The visual inspection of generated state machines would yield a form of equivalence queries. Otherwise, we might investigate ways of mitigating the lack of user control by injecting further domain knowledge or automating the oracle (see e.g., [Wal07]).

Lack of knowledge and classification errors

One difficulty with incremental system modeling is related to the lack of system knowledge. When using the interactive feature of QSM for example, the user might be faced with a scenario question for which she does not have a clear classification answer. In such situation, “don’t know” answers are worth investigating. Another approach would consist of allowing the end-user to defer some scenario questions so that they can be answered later as more system knowledge is being gained.

An open issue related to the previous one is the robustness against possible misclassification of scenario questions by the end-user. Traditional ways of dealing with noisy inputs include probabilistic learning methods. Such methods could be adapted in our context, especially in presence of multiple stakeholders.

Multi-view modeling actually provides another way of dealing with classification errors. For example, domain knowledge and goals may help detecting and/or correcting such mistakes. Indeed, misclassification typically leads to consistency checks failing if domain properties and goals are available. Those failures can be effectively fixed provided they are detected soon enough. Once again, a better integration of model analysis and model synthesis techniques might prove very useful here.

Implied scenarios

The potential nuisance of implied scenarios has been discussed in Section 4.6. Implied scenarios occur when a distributed system is designed globally while implemented component-wise. In our framework, problematic implied scenarios are system behaviors correctly rejected by the system synthesized globally, as required by a goal for instance, but exhibited in the system when individual agent state machines are composed.

Existing techniques, such as [Uch04], might be adapted to our framework in order to detect implied scenarios and submit them as additional scenario questions to the end-user (see Section 8.2.3). Even with the availability of such a detection technique, implied scenarios still raise important open issues.

- The presence of negative implied scenarios requires refactoring of the system decomposition and agent interfaces. An important issue with implied scenarios is the lack of systematic guidance and scenario refactoring techniques to eliminate them.

- Another important problem occurs when we don't know whether an implied scenario should be accepted or rejected. This is related to the problem of lack of knowledge previously discussed. Implied scenarios whose status is unclear stay potentially harmful until a decision is made about them.

- The late discovery of harmful implied scenarios may be due to an inadequate use of the multi-view modeling framework or to weaknesses of the formal framework itself. For example, our definition of negative scenario borrowed from [Uch02] should probably be revisited. Indeed, such scenarios define negative system traces only; in particular, they

do not define a negative trace in the state machine of the agent that sends the proscribed event. In other words, negative scenarios do not capture a restriction on the behavior of a single agent. Similarly, goals are considered system-wide. Considering only goals that are realizable by at least one agent is often sufficient to avoid related implied scenario problems in the first place.

While those restrictions and open issues might limit the applicability of our techniques in specific cases, they also pave the way for numerous improvements and research in multi-view model synthesis.

Bibliography

[Sa06] Sauer, T. and Maximini, K. Using workflow context for automated enactment state tracking. In Mirjam Minor, editor, *Workshop Proceedings: 8th European Conference on Case-Based Reasoning, ldeniz/Fethiye, Turkey, Workshop: "Case-based Reasoning and Context Awareness (CACOA 2006)"*, pages 300–314. Universitt Trier, Department of Business Information Systems II, September 2006.

[Adr06] Adriaans, P. and Jacobs, C. Using MDL for Grammar Induction. *Grammatical Inference: Algorithms and Applications*, pages 293–306, 2006.

[Aho86] Aho, A.V. and Sethi, R. and Ullman, J.D. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Alp86] Alpern, B. and Schneider, F.B. Recognizing safety and liveness. Technical report, Ithaca, NY, USA, 1986.

[Alr07] Alrajeh, D. and Ray, O. and Russo, A. and Uchitel, S. Inductive logic programming. chapter Extracting Requirements from Scenarios with ILP, pages 64–78. Springer-Verlag, Berlin, Heidelberg, 2007.

[Alu99] Alur, R. and Yannakakis, M. Model checking of message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pages 114–129, London, UK, 1999. Springer-Verlag.

[Alu00] Alur, R. and Etessami, K. and Yannakakis, M. Inference of message sequence charts. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 304–313, New York, NY, USA, 2000. ACM.

[Ang78] Angluin, D. On the complexity of minimum inference of regular sets. *Information and Control*, 39:337–350, 1978.

[Ang87] Angluin, D. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[Asa95] Asarin, E. and Maler, O. and Pnueli, A. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20, London, UK, 1995. Springer-Verlag.

[Bal85] Balzer, R. A 15 year perspective on automatic programming. *IEEE Trans. Softw. Eng.*, 11:1257–1268, November 1985.

[Bon05] Bongard, J. and Lipson, H. and Wrobel, S. Active coevolutionary learning of deterministic finite automata. *Journal of Machine Learning Research*, 6, 2005.

[Bri03] Briand, L.C. and Labiche, Y. and Miao, Y. Towards the reverse engineering of uml sequence diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 57–, Washington, DC, USA, 2003. IEEE Computer Society.

[Bro87] Brooks, F.P. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[Bry86] Bryant, R.E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[Buh05] Buhler, P.A. and Vidal, J.M. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management*, 6:61–87, 2005. 10.1007/s10799-004-7775-2.

[Cla82] Clarke, E.M. and Emerson, E.A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[Cla89] Clarke, E. and Long, D. and McMillan, K. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[Cla08] Clarke, L.A. and Avrunin, G.S. and Osterweil, L.J. Using software engineering technology to improve the quality of medical processes. In *International Conference on Software Engineering*, pages 889–898, 2008.

[Com11] Combefis, S. and Giannakopoulou, D. and Pecheur, C. and Feary, M. A formal framework for design and analysis of human-machine interaction. In *2011 IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 2011.

[Cos97] Coste, F. and Nicolas, J. Regular inference as a graph coloring problem. In *In Workshop on Grammar Inference, Automata Induction, and Language Acquisition (ICML' 97*, pages 9–7, 1997.

[Cos98] Coste, F. and Nicolas, J. How considering incompatible state mergings may reduce the DFA induction search tree. In *Grammatical Inference, ICGI'98*, number 1433 in Lecture Notes in Artificial Intelligence, pages 199–210, Ames, Iowa, 1998. Springer Verlag.

[Cos04] Coste, F. and Fredouille, D. and Kermorvant, C. and de la Higuera, C. Introducing domain and typing bias in automata inference. In *Grammatical Inference: Algorithms and Applications*, number 3264 in Lecture Notes in Artificial Intelligence, pages 115–126, Athens, Greece, 2004. Springer Verlag.

[Dam01] Damm, W. and Harel, D. Lscs: Breathing life into message sequence charts. *Form. Methods Syst. Des.*, 19:45–80, July 2001.

[Dam05] Damas, C. and Lambeau, B. and Dupont, P. and van Lamsweerde, A. . Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[Dam06] Damas, C. and Lambeau, B. and van Lamsweerde, A. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *International ACM Symposium on the Foundations of Software Engineering*, pages 197–207, Portland, Oregon, November 2006.

[Dam09] Damas, C. and Lambeau, B. and Roucoux, F. and van Lamsweerde, A. Analyzing critical process models through behavior model synthesis. In *ICSE'09: 31th International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[Dam10] Damas, C. and Lambeau, B. and Roucoux, F. and van Lamsweerde, A. Abstractions for analyzing decision-based process models. Technical report, Computing Science Department, Université catholique de Louvain, May 2010.

[Dam11] Damas, C. *Analyzing Multi-View Models of Software Systems*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, 2011.

[Dat97] Date, C.J. and Darwen, H. *A Guide to SQL Standard*. Addison-Wesley, 4th edition edition, 1997.

[Dat03] Date, C.J. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition, 2003.

[Dup94] Dupont, P. and Miclet, L. and Vidal, E. What is the search space of the regular inference ? In *Grammatical Inference and Applications, ICGI'94*, number 862 in Lecture Notes in Artificial Intelligence, pages 25–37, Alicante, Spain, 1994. Springer Verlag.

[Dup96] Dupont, P. Incremental regular inference. In *Grammatical Inference : Learning Syntax from Sentences, ICGI'96*, number 1147 in Lecture Notes in Artificial Intelligence, pages 222–237. Springer Verlag, 1996.

[Dup08] Dupont, P. and Lambeau, B. and Damas, C. and van Lamsweerde, A. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22:77–115, 2008.

[Dvt05] Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede, editors. *Process-Aware Information Systems : Bridging People and Software through Process Technology.* Wiley-Interscience, Hoboken, N.J., 2005.

[Eis02] Eisenberg, J.D. *SVG Essentials.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1 edition, 2002.

[Eng85] Engelfriet J. Determinacy -¿ (observation equivalence = trace equivalence). *Theor. Comput. Sci.*, 36:21–25, 1985.

[Esh06] Eshuis, R. Symbolic model checking of uml activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15:1–38, January 2006.

[Fea87] Feather, M.S. Language support for the specification and development of composite systems. *ACM Trans. Program. Lang. Syst.*, 9:198–234, March 1987.

[Fea97] Feather, M.S. and Fickas, S. and Finkelstein, A. and van Lamsweerde, A. Requirements and specification exemplars. *Automated Software Engineering*, 4:419–438, 1997.

[Fin92] Finkelstein, A. and Kramer, J. and Nuseibeh, B. and Finkelstein, L. and Goedicke, M. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. 2:31–57+, 1992.

[Gam03] Gamma, E. and Beck, K. *Contributing to Eclipse: Principles, Patterns, and Plugins.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[Gar10] García, P. and de Parga, M.V. and López, D. and Ruiz, J. Learning automata teams. In *Proceedings of the 10th international colloquium conference on Grammatical inference: theoretical results and applications*, ICGI'10, pages 52–65, Berlin, Heidelberg, 2010. Springer-Verlag.

[Gia99] Giannakopoulou, D. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College London, London, 1999.

[Gia02] Giannakopoulou, D. and Lerda, F. From states to transitions: Improving translation of ltl formulae to büchi automata. In *Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, FORTE '02, pages 308–326, London, UK, UK, 2002. Springer-Verlag.

[Gia03] Giannakopoulou, D. and Magee, J. Fluent model checking for event-based systems. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266, New York, NY, USA, 2003. ACM.

[Gol67] Gold, E.M. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[Gol78] Gold, E.M. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[Gra09] Grando, M.A. and Glasspool, D.W. and Fox, J. Petri nets as a formalism for comparing expressiveness of workflow-based clinical guideline languages. In Wil Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, Clemens Szyperski, Danilo Ardagna, Massimo Mecella, and Jian Yang, editors, *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing*, pages 348–360. Springer Berlin Heidelberg, 2009.

[Hal04] Hall, R.J. and Zisman, A. Omml: A behavioural model interchange format. *Requirements Engineering, IEEE International Conference on*, 0:272–282, 2004.

[Har03a] Harel, D. and Marelly, R. *Come, let's play: scenario-based programming using LSCs and the play-engine*. Number v. 1. Springer, 2003.

[Har03b] Harel, D. and Marelly, R. Specifying and executing behavioral requirements: the play-in/play-out approach. 2:82–107+, 2003.

[Har05] Harel, D. and Kugler, H. and Pnueli, A. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In *Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday*,

pages 309–324. Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, Springer, 2005.

[Hen00] Henriksen, J.G. and Mukund, M. and Kumar, K.N. and Thiagarajan, P.S. On message sequence graphs and finitely generated regular msc languages. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, ICALP '00, pages 675–686, London, UK, 2000. Springer-Verlag.

[Heu10] Heule, M.J.H. and Verwer, S. Exact dfa identification using sat solvers. In *Proceedings of the 10th international colloquium conference on Grammatical inference: theoretical results and applications*, ICGI'10, pages 66–79, Berlin, Heidelberg, 2010. Springer-Verlag.

[Hoa85] Hoare, C.A.R. *Communicating Sequential Processes (Prentice Hall International Series in Computing Science)*. Prentice Hall, April 1985.

[Hol97] Holzmann, G.J. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[Hop79] Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, Massachusetts, 1979.

[ITU96] ITU. Message sequence charts, recommandation z.120. *International Telecom Union, Telecommunication Standardization Sector*, 1996.

[Jac95] Jackson, M. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 283–292, New York, NY, USA, 1995. ACM.

[Jos96] Joseph, M. *Real-Time Systems: Specification, Verification and Analysis*. Prentice-Hall, 1996.

[Kar00] Karamanolis, C.T. and Giannakopoulou, D. and Magee, J. and Wheater, S.M. Model checking of workflow schemas. In *Proceedings of the 4th International conference on Enterprise Distributed Object Computing*, EDOC '00, pages 170–181, Washington, DC, USA, 2000. IEEE Computer Society.

[Kel76] Keller, R.M. Formal verification of parallel programs. *Communications of the ACM*, 19:371–384, July 1976.

[Koh00] Kohler, H.J. and Nickel, U. and Niere, J. and Zundorf, A. Integrating uml diagrams for production control systems. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 241–251, New York, NY, USA, 2000. ACM.

[Kru00] Kruger, I.H. *Distributed System Design with Message Sequence Charts.* Dissertation, Technische Universitt Mnchen, Mnchen, 2000.

[Lam94] Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, May 1994.

[Lam08] Lambeau, B. and Damas, C. and Dupont, P. State-merging dfa induction algorithms with mandatory merge constraints. In *ICGI '08: Proceedings of the 9th international colloquium on Grammatical Inference*, pages 139–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[Lan92] Lang, K.J. Random dfa's can be approximately learned from sparse uniform examples. In *Proceedings of the fifth annual workshop on Computational learning theory*, COLT '92, pages 45–52, New York, NY, USA, 1992. ACM.

[Lan98] Lang, K.J. and Pearlmutter, B.A. and Price, R.A. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Grammatical Inference*, number 1433 in Lecture Notes in Artificial Intelligence, pages 1–12, Ames, Iowa, 1998. Springer-Verlag.

[Lar88] Larsen, K.G. and Thomsen, B. A modal process logic. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science (LICS 1988)*, pages 203–210. IEEE Computer Society Press, July 1988.

[Ler04] Lerner, B.S. Verifying process models built using parameterized state machines. *SIGSOFT Softw. Eng. Notes*, 29:274–284, July 2004.

[Les07] Leskovec, J. and Kleinberg, J. and Faloutsos, C. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2, 2007.

[Let02] Letier, E. and van Lamsweerde, A. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 83–93, New York, NY, USA, 2002. ACM.

[Let05] Letier, E. and Kramer, J. and Magee, J. and Uchitel, S. Fluent temporal logic for discrete-time event-based models. *SIGSOFT Softw. Eng. Notes*, 30:70–79, September 2005.

[Let08] Letier, E. and Kramer, J. and Magee, J. and Uchitel, S. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engg.*, 15:175–206, June 2008.

[Lia06] Liang, H. and Dingel, J. and Diskin, Z. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, SCESM '06, pages 5–12, New York, NY, USA, 2006. ACM.

[Luc03] Lucas, S.M. and Reynolds, T.J. Learning dfa: evolution versus evidence driven state merging. In *IEEE Congress on Evolutionary Computation (1)*, pages 351–358, 2003.

[Luc05] Lucas, S.M. and Reynolds, T.J. Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:1063–1074, 2005.

[Mag95] Magee, J. and Dulay, N. and Eisenbach, S. and Kramer, J. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[Mag97] Magee, J. and Kramer, J. and Giannakopoulou, D. Analysing the behaviour of distributed software architectures: a case study. In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems*, FTDCS '97, pages 240–, Washington, DC, USA, 1997. IEEE Computer Society.

[Mak01] Makinen, M. and Systä, T. MAS – an interactive synthetiser to support behavorial modeling in UML. In *27th International Conference on Software Engineering*, 2001.

[Man71] Manna, Z. and Waldinger, R.J. Toward automatic program synthesis. *Commun. ACM*, 14:151–165, March 1971.

[Man92] Manna, Z. and Pnueli, A. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[Man02] Manolescu, D.A. Workflow enactment with continuation and future objects. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and*

applications, OOPSLA '02, pages 40–51, New York, NY, USA, 2002. ACM.

[Mar10] Martin, F. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[Mid00] Middleton, S. and Roberts, A. *Integrated care pathways: a practical approach to implementation*. Butterworth-Heinemann, 2000.

[Mil89] Milner, R. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Mil02] Miller, R. and Shanahan, M. Some alternative formulations of the event calculus. In *Computer Science; Computational Logic; Logic programming and Beyond*, pages 452–490. Springer, 2002.

[Mit80] Mitchell, T.M. The Need for Biases in Learning Generalizations. Technical report, Rutgers Computer Science Department, May 1980.

[MK99] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999.

[Mue00] Muehlen, M. and Rosemann, M. Workflow-based process monitoring and controlling ¾ technical and organizational issues. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 6 - Volume 6*, HICSS '00, pages 6032–, Washington, DC, USA, 2000. IEEE Computer Society.

[Myl92] Mylopoulos, J. and Chung, L. and Nixon, B. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Softw. Eng.*, 18:483–497, June 1992.

[Obj04] Object Management Group. The uml 2.0 superstructure specification. Specification Version 2, Object Management Group, 2004.

[Obj08] Object Management Group. Business Process Modeling Notation, v1.1. Technical report, Object Management Group, 2008.

[Onc92] Oncina, J. and García, P. Inferring regular languages in polynomial update time. In N. Pérez de la Blanca, A. Sanfeliu, and E.Vidal, editors, *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.

[Onc93] Oncina, J. and García, P. and Vidal E. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):448–458, 1993.

[Par81] Park, D. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[Pil10] Pilgrim, M. *HTML5 - Up and Running: Dive Into the Future of Web Development.* O'Reilly, 2010.

[Pnu89] Pnueli, A. and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[Rum91] Rumbaugh, J. and Blaha, M. and Premerlani, W. and Eddy, F. and Lorenson, W. *Object-Oriented Modeling and Design.* Prentice Hall, Inc., 1st edition, 1991.

[Tra73] Trakhtenbrot, B. and Barzdin, Y. *Finite Automata: Behavior and Synthesis.* 1973.

[Uch01a] Uchitel, S. and Kramer, J. A workbench for synthesising behaviour models from scenarios. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 188–197, Washington, DC, USA, 2001. IEEE Computer Society.

[Uch01b] Uchitel, S. and Kramer, J. and Magee, J. Detecting implied scenarios in message sequence chart specifications. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 74–82, New York, NY, USA, 2001. ACM.

[Uch02] Uchitel, S. and Kramer, J. and Magee, J. Negative scenarios for implied scenario elicitation. *SIGSOFT Softw. Eng. Notes*, 27:109–118, November 2002.

[Uch03] Uchitel, S. and Kramer, J. and Magee, J. Synthesis of behavorial models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[Uch04] Uchitel, S. and Kramer, J. and Magee, J. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13:37–85, January 2004.

[Uch07] Uchitel, S. and Brunet, G. and Chechik, M. Behaviour model synthesis from properties and scenarios. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 34–43, Washington, DC, USA, 2007. IEEE Computer Society.

[Uch09] Uchitel, S. and Brunet, G. and Chechik, M. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Softw. Eng.*, 35:384–406, May 2009.

[Val84] Valiant, L.G. A theory of the learnable. *Commun. ACM*, 27:1134–1142, November 1984.

[van98] van Lamsweerde, A. and Willemet, L. Inferring declarative requirements specifications from operational scenarios. *Software Engineering, IEEE Transactions on*, 24(12):1089–1114, 1998.

[van00a] van Deursen, A. and Klint, P. and Visser, J. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000.

[van00b] van Lamsweerde, A. Requirements engineering in the year 00: a research perspective. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 5–19, New York, NY, USA, 2000. ACM.

[van04] van Lamsweerde, A. Goal-oriented requirements enginering: A roundtrip from research to practice. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International*, pages 4–7, Washington, DC, USA, 2004. IEEE Computer Society.

[Van05] Vanderaalst, W. and Terhofstede, A. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, June 2005.

[van09] van Lamsweerde, A. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, March 2009.

[Wal07] Walkinshaw, N. and Bogdanov, K. and Holcombe, M. and Salahuddin, S. Reverse engineering state machines by interactive grammar inference. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 209–218, Washington, DC, USA, 2007. IEEE Computer Society.

[Wal08] Walkinshaw, N. and Bogdanov, K. and Johnson, K. Evaluation and comparison of inferred regular grammars. In *Proceedings of the 9th international colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '08, pages 252–265, Berlin, Heidelberg, 2008. Springer-Verlag.

[War85] Ward, P.T. and Mellor, S.J. *Structured Development for Real-Time Systems*. Prentice Hall Professional Technical Reference, 1985.

[Was03] Wasowski, A. On efficient program synthesis from statecharts. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 163–170, New York, NY, USA, 2003. ACM.

[Wei98] Weidenhaupt, K. and Pohl, K. and Jarke, M. and Haumer, P. Scenarios in system development: current practice. *Software, IEEE*, 15(2):34–45, 1998.

[Whi00] Whittle, J. and Schumann, J. Generating statechart designs from scenarios. In *ICSE*, pages 314–323, 2000.

[Yu,93] Yu, E.S.K. Modeling organizations for information systems requirements engineering. In *Proceedings of the Requirements Engineering Conference, 1st IEEE International*, pages 34–41, Washington, DC, USA, 1993. IEEE Computer Society.

[Yu,05] Yu, Y. and Wang, Y. and Mylopoulos, J. and Liaskos, S. and Lapouchnian, A. and Prado Leite, J.C.S. Reverse engineering goal models from legacy code. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 363–372, Washington, DC, USA, 2005. IEEE Computer Society.