Imperial College London

Department of Computing

Final Year Project Report

Refining Labelled Transition Systems Using Scenario-Based Specifications

Author:
Diana Ramchandani

Project Supervisor:
Dr Alessandra Russo

Project Co-Supervisor:
Dalal Alrajeh

June 25, 2009

Acknowledgements

Firstly, I would like to give special thanks to my supervisors, Dr Alessandra Russo and Dalal Alrajeh, for their diligent help, support and encouragement throughout the entire course of the project. The critical review and guidelines they provided me with at every meeting as well as the brainstorming sessions helped me explore a myriad of different alternatives during the development phase and to shape the techniques used, thus improving my apprehension of the various concepts. On a more personal level, I really appreciate their optimism, enthusiasm and generosity.

Secondly, I would like to thank Dr Emil Lupu for agreeing to be my second marker, and for his invaluable feedback, guidance and suggestions during early stages of the project.

I would also like to thank Will Heaven for contributing a significant part of his precious time to help me better my understanding of the back-end to the existing LTSA tool, hence providing me with an insight to some of the design issues during the implementation of the refinement algorithm.

In addition, I would like to thank my friends and family for their endless support, patience, and cooperation throughout my time at university.

Abstract

Labelled Transition Systems(LTSs) are often used in theoretical Computer Science in order to study computations through the analysis of the system's states and transitions. Large and complex system models can be constructed by the composition of multiple LTSs, and the LTSA can be used as a general purpose tool to explore such systems through a number of different perspectives. Amongst these, it is necessary to verify that a system satisfies a set of defined properties. Current approaches perform similar checking, but often they require constant involvement from the user, and the algorithms rely on various kinds of pre-processing of the input.

The aim of this project is to ensure that a system specification adheres to examples of desirable and undesirable system behaviour provided by end-users, and consequently to refine the initial LTS directly such that the resulting model is consistent with user-provided scenarios.

More specifically, the objective is to write and implement an algorithm that takes an initial transition system as input together with sets of positive and negative scenarios, and outputs a refined model if one exists. We also include a simple addition to the existing LTSA architecture to demonstrate functionality of this refinement feature.

Contents

Chapter 1

Introduction

1.1 Motivation

Labelled Transition Systems (LTS), are operational models of system behaviours that can be analysed in various ways with respect to given safety properties of the system. However, they often give a holistic view of the system, thereby also covering behaviour that is undesirable according to the system specification. It is therefore in the interest of requirements engineers to refine these LTS models so that they can provide a more synthesised view of the system using scenarios and knowledge about the system domain.

We may want to initiate the refinement starting from the most general system which has no specification associated with it, and is defined by domain knowledge alone. However, systems can also be synthesised at later stages of the development process as a form of validation with respect to certain properties. Sets of scenarios and counter-examples are provided by the user or generated as a result of the validation process, which are taken into account in order to generate a more refined LTS. Our motivation behind the LTS refinement approach presented in this report comes from the desire to solve the problem of elaborating behavioural models from scenarios and knowledge about the system domain, by means of subsequent stages of refinement directly applied to the given LTS model, as shown in Figure 1.1.

Scenario-based specifications are becoming increasingly popular as part of the requirements engineering process. They are partial operational descriptions of system behaviours, and as such are very effective for communicating with the users. Initially we assume that the given sets of positive and negative scenarios are covered by the LTS that we start from, and this needs to be verified at every subsequent refinement stage.

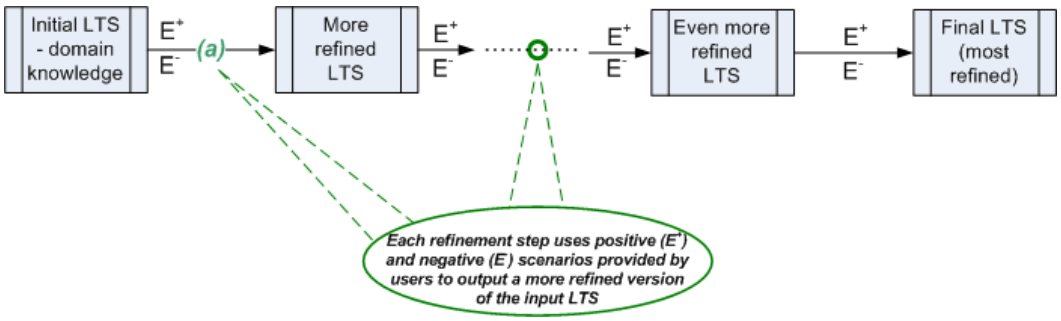

Figure 1.1: Synthesis of LTS models

Recent work has shown how an appropriate inductive learning algorithm can help support the process of refining end-user goals into operational requirements via scenarios, so that appropriate system models can be constructed in line with the specification. However, this requires "encoding" of the LTSs into specific types of logic programs and "decoding" of the learned clauses into refined LTSs that accept (resp. reject) new given positive (resp. negative) scenarios.

Stakeholders (users or beneficiaries) often provide partial specifications of systems, from which LTS models are generated. These models may include an excess number of states and hence may cover a number of unwanted scenarios if the system specification is not sufficiently constrained. We intend to synthesise the most general LTS (for which there is *no* specification yet, only domain knowledge) according to sets of positive and negative scenarios covered by the system in question, so that the resulting model is a *refined* version of the initial LTS, which still accepts the desirable system behaviour and simultaneously rejects *all* the given negative scenarios. This is denoted by the edge labelled *(a)* in Figure 1.1, and its flow is illustrated in Figure 1.2.

System behaviour and properties are specified using *Finite State Process*(FSP) notation in the LTSA (Section 2.1.1), which in addition provides methods for checking these properties, as described in Section 2.1. Even though our approach does not focus on checking such properties, it forms part of the synthesis process, which verifies their satisfaction when implicitly expressed in the given scenarios. The following example helps clarify these concepts.

Let us consider a train system, which can move, stop, have its alarm button pressed, and have its doors opened or closed. An FSP specification for this system that includes mainly domain conditions would enable sequences

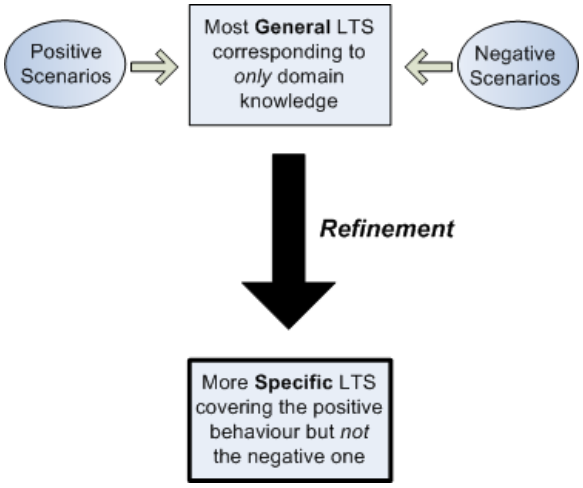

Figure 1.2: Diagrammatic Representation of Our Aim

of these actions to occur without taking into account some of the system's goals. If the system requires all doors to be closed while the train is in motion due to passenger safety reasons (i.e. a safety property for the train system), then this would not necessarily be captured by the LTS resulting from the initial FSP. Our intention is to take in this sort of additional pre-condition information as input, in the form of examples of behaviour that the system should always (resp. never) be able to exhibit, and consequently restrict the set of actions possible in order to satisfy the requirements provided. In this case, if a passenger presses the alarm button in an emergency, the train would firstly need to come to a halt, and only then can its doors be opened. Therefore, our algorithm would work on the initial LTS, and output a resulting LTS which ensures that this is adhered to.

This report presents a way of developing an algorithm which analyses LTS models with respect to positive and negative scenarios and defines refinement operators directly on the LTS models that would simulate the generalisation process of an inductive computation. The outcomes of the algorithm explained herewith are therefore evaluated by comparing its results to those obtained by the ILP task when applied to a specific case study, as they both share similar aims.

1.2 Contributions

The principal contribution of this project has been the development of an algorithm that can be applied directly to Labelled Transition Systems in order to refine them according to given sets of positive and negative scenarios, representing desirable and undesirable system behaviour, respectively. The aim is to enable the system to exhibit all the positive behaviour, whilst at the same time to disallow it from engaging in any actions or sequences of actions that could lead to unwanted behaviour. The relevant background material can be studied in Chapter 2.

As part of this holistic approach, we have investigated various ways in which the positive scenarios provided can be synthesised and deployed together with the negative scenarios, to prevent the obtention of an overly constrained model whilst preserving as much of the desirable system behaviour as possible. Chapter 3 includes details regarding these related approaches.

We have implemented the proposed algorithm within the existing LTSA tool developed at Imperial College London using Java, mainly because the LTSA is written in this language as well. Thus, we took advantage of a relatively quick and easy means of integrating the new code into the existing one. Further design and implementation details are discussed in Chapter 4.

We assume that, together with an initial set of requirements or other conditions that may have been included in the system specification, the input to our algorithm will be the most generalised model corresponding to the system. We can then take into account sets of positive and negative scenarios that are covered by the system in question, respectively, rather than relying on arbitrary ones. The LTS model given in Figure 1.3 on Page 7 is an example of an initial model containing the maximum number of states possible, and hence exhibiting the maximum behaviour.

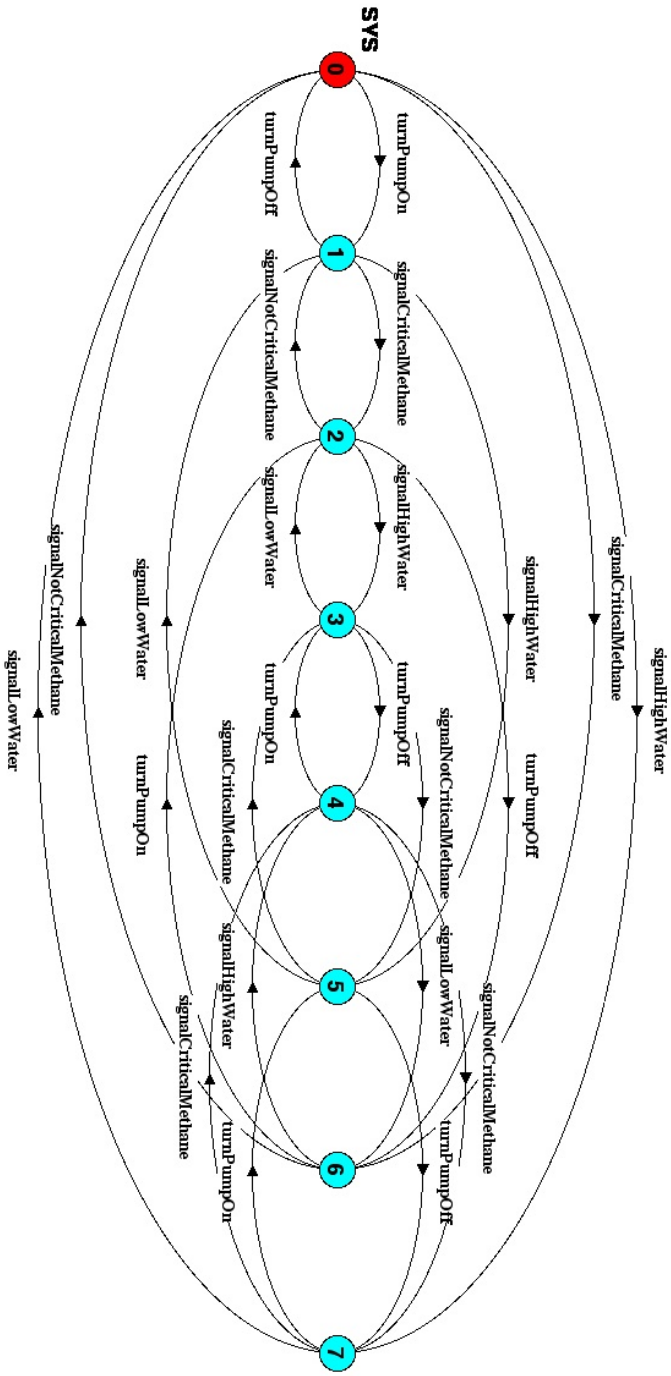

Figure 1.3: An example LTS for a Simple Mine Pump

Chapter 2

Background

2.1 Labelled Transition Systems (LTS)

Labelled Transition Systems (LTSs) belong to a specific class of automata, called the *Finite State Automata* (FSA). A *finite state automaton* is a 5-tuple (Q,Σ,δ,q_0,F), where Q is a finite set of states (q_0 being the initial state where all possible paths start from), Σ is a set of event labels that are used to identify transitions between states, δ is a transition function mapping $Q \times \Sigma$ to 2^Q, and F is a subset of Q representing just the *accepting* states, which are the last states of all paths that are accepted by the automaton. An example FSA is shown in Figure 2.1, where the transition labelled ϵ implies that it can be traversed without an event occurring, and final states are represented by two concentric circles. We can formally define it as the 5-tuple (Q,Σ,δ,q_0,F), where the set of states Q in this case is {A, B, C, D, E, F}, the initial state q_0 is A, the set of final states F is {A, C, D}, the set of event labels Σ is {a, b, c, d, e, f, ϵ}, and the elements (A, a, B), (A, a, C), (A, b, F), (B, c, D), (B, d, E), (C, e, D), (C, f, E), (F, ϵ, B), and (F, a, E) are those defined by the transition function δ.

A *deterministic finite-state automaton* (DFA) is present if, for any q in Q and any e in Σ, $\delta(q,e)$ has at most one member. In addition, a DFA has only one initial state, and no ϵ transitions. The corresponding DFA for the FSA in Figure 2.1 is shown in Figure 2.2 on page 9. We can see that the ϵ transition that previously existed between states F and B has now been removed, and the multiple outgoing 'a' transitions from state A are now just reduced to a single outgoing transition with this label.

An LTS only includes accepting states, and so the sets Q and F are the same. A *Labelled Transition System M* can therefore be defined as in

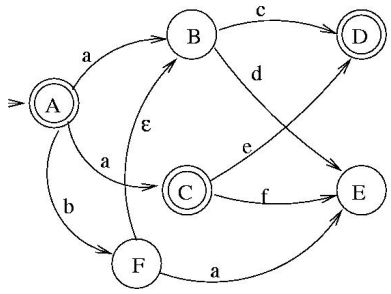

Figure 2.1: An example FSA [1]

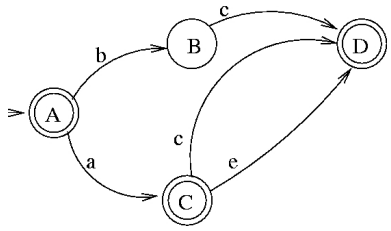

Figure 2.2: An example DFA for the FSA in Figure 2.1 [1]

[8, 9, 16], as a 4-tuple (Q,A,δ,q_0), where Q is a finite set of states, $A \subseteq Act$ is the countable set of observable actions called the communicating *alphabet* of M, $\delta \subseteq Q \times (A \cup \{\tau\}) \times Q$ is a labelled transition relation, and q_0 is the initial state. τ denotes a local action that is *unobservable* or *invisible*, and so $\tau \notin Act$. If a state q in the LTS has no outgoing transition with label l, then no action with label l can take place when the system is in state q. We use π to denote a special *error state*, which models the fact that a safety property has been violated by the system. It is required that the error state has no outgoing transitions, and the corresponding LTS can be denoted as $\Pi = \langle \{\pi\}, Act, \emptyset, \pi \rangle$. One thing to note is that the initial state of all LTSs in this report is assumed to be state 0.

One of the differences between LTSs and finite state automata lies in the finiteness of the number of states and transitions. The number of states and transitions in an LTS need not be finite, or even countable. Each state in an LTS is considered as an *accepting* state, whilst FSAs can only have a finite number of final states, which are represented by concentric circles, and are used for recognising a given set of finite strings.

LTSs are graph-like structures that show the different states that a system can be in, and possible transitions between them [13, 23, 33]. Each tran-

sition is labelled by an action or event. An example of an LTS diagram for a simple mine pump control system described in [2, 3] is shown in Figure 1.3 on Page 7, where the finite set of states Q is {0, 1, 2, 3, 4, 5, 6, 7}, the set of observable actions A is {*turnPumpOn, turnPumpOff, signalHighWater, signalLowWater, signalCriticalMethane, signalNotCriticalMethane*}, and the labelled transition relation δ is satisfied by elements including (0, *signalHighWater*, 0), (0, *signalNotCriticalMethane*, 1) and (6, *turnPumpOn*, 4), the initial state q_0 being state 0 in this case.

Traces

An *execution* or a *trace* of an LTS M is a sequence of observable or τ actions that M can perform, starting at its initial state, i.e. q_0. It is described as $M = \langle Q,\ A,\ \delta, q_0\rangle$ transiting to $M' = \langle Q,\ A,\ \delta, q_0'\rangle$ with action a, denoted as $M \xrightarrow{a} M'$, iff $(q_0, a, q_0') \in \delta$. Even though all states in an LTS are accepting states, we can define the set of states accepting a particular action a as $\{q_0' \mid (q_0,a,q_0') \in \delta\}$. We could, therefore, use $\delta(q_0,a)$ to give us the state that M would be in after executing action a in the state q_0. The set of all traces of M is called the *language* of M, denoted *L(M)*.

Example traces in the LTS diagram in Figure 1.3 include:

$$\mathbf{0} \xrightarrow{signalLowWater} \mathbf{4} \xrightarrow{signalhighWater} \mathbf{0}$$

$$\mathbf{0} \xrightarrow{turnPumpOff} \mathbf{2} \xrightarrow{signalHighWater} \mathbf{2} \xrightarrow{turnPumpOn} \mathbf{0}$$

$$\mathbf{0} \xrightarrow{signalNotCriticalMethane} \mathbf{1} \xrightarrow{turnPumpOff} \mathbf{3} \xrightarrow{turnPumpOn} \mathbf{1} \xrightarrow{signalCriticalMethane} \mathbf{0}$$

These can also be represented as ⟨*signalLowWater, signalHighWater*⟩, ⟨*turnPumpOff, signalHighWater, turnPumpOn*⟩ and ⟨*signalNotCriticalMethane, turnPumpOff, turnPumpOn, signalCriticalMethane*⟩.

We define a *finite execution* of an LTS M (Q,A,δ,q_0) as a finite sequence of consecutive transitions $\langle a_1,\ldots,a_n\rangle$ with $a_i \in A$, accepted by the LTS from its initial state, i.e. q_0. As in [5], for any natural number $n \in \mathbb{N}$, states $s_i \in S$ (a finite set of states) and actions $a_i \in A$, with $i \in \mathbb{N}$ and $0 \leq i < n$, $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots\ s_{n-1} \xrightarrow{a_{n-1}} s_n$ is called an execution sequence of length n of M iff $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \in \mathbb{N}$ with $0 \leq i < n$. We can say that state s_n is *reachable* from state s_0, because any state in an LTS is *reachable* in M iff it is reachable from q_0. This does not necessarily mean that a single transition

has to exist between the reachable state and q_0, as the state will still be reachable even if this is achieved through a set of transitions which start at state 0 and cover additional states during the trace.

Composition

It is often the case where agents responsible for particular system behaviour are each specified by a separate component, in order to clearly represent each individual functionality. When it comes to larger systems that involve the interaction between these agents, the individual behaviours are combined into a single model. The scope of this kind of systems that can be modelled using LTSs can be very large and complex, so different processes are often composed using the *parallel composition* operation "||", which results in a composite process. The term *process* is used in the context of LTSs to denote a *Finite State Process* (FSP). If P and Q are processes, then $(P \mid\mid Q)$ represents the concurrent execution of P and Q. This is a commutative and associative operator, which means:

- $(P \mid\mid Q) = (Q \mid\mid P)$
- $(P \mid\mid Q) \mid\mid R = P \mid\mid (Q \mid\mid R)$

Thus, the behaviour of several LTSs can be combined because the composite process allows the visible actions of processes to be *shared* (those that are common to their alphabets). A shared action must be executed at the same time by all processes that participate in the shared action, whilst unshared actions can be interleaved arbitrarily, hence causing asynchronous behaviour of the composed model. If $M_1 = \langle Q^1,\ A^1, \delta^1,\ q_0^1 \rangle$ and $M_2 = \langle Q^2, A^2, \delta^2, q_0^2 \rangle$, then, if $M_1 = \Pi$ or $M_2 = \Pi$, $M_1 \mid\mid M_2 = \Pi$. Otherwise, $M_1 \mid\mid M_2 = \langle Q,\ A, \delta, q_0 \rangle$ where $Q = Q^1 \times Q^2$, $A = A^1 \cup A^2$, $q_0 = (q_0^1,\ q_0^2)$, and δ is defined as follows, where a can be an observable action or τ :

- $\frac{M_1 \xrightarrow{a} M_1', a \notin A^2}{M_1 \mid\mid M_2 \xrightarrow{a} M_1' \mid\mid M_2}$
- $\frac{M_1 \xrightarrow{a} M_1', M_2 \xrightarrow{a} M_2', a \neq \tau}{M_1 \mid\mid M_2 \xrightarrow{a} M_1' \mid\mid M_2'}$

An example consisting of two LTSs is shown in Figures 2.3 and 2.4 which represent a Simple Mine Pump and its Water Sensor, respectively, together with the result of their composition in Figure 2.5 [23]. Note that the synchronised actions are *turnPumpOn* and *turnPumpOff*, as these are

shared between the processes, and therefore the pump cannot execute a *turnPumpOn* action because it needs to wait for a signal of *highWater* from the water sensor. Similarly, the pump cannot perform the *turnPumpOff* action because it needs to wait for the sensor to signal *lowWater* before doing so.

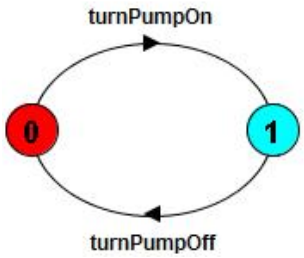

Figure 2.3: LTS 1

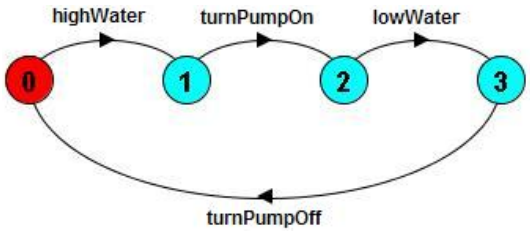

Figure 2.4: LTS 2

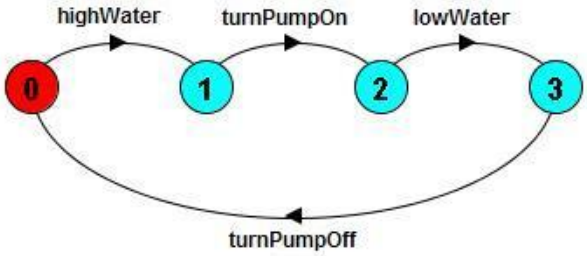

Figure 2.5: Composed LTS from Figures 2.3 and 2.4

Properties

A *safety LTS* is a deterministic LTS that contains no π states, and is used to specify a *safety property*. This LTS's language will then define the set of acceptable behaviours possible over the set of available actions. Another LTS is then said to satisfy the safety LTS iff any trace in the LTS is also a trace in the safety LTS, given its set of actions.

An *error LTS* is also available to check properties, and trap possible violations using the π state. Every state other than the error state will have outgoing transitions, hence making the error LTS complete. Detection of violations of a property is possible via the parallel composition of the LTS being tested, with the error LTS corresponding to the property being

checked. If the π state is reachable in this composition, then the LTS is said to violate the property.

If we regard an LTS as a component of a larger specification, which contains all the states a component may reach and all the transitions it may perform, we can use the Labelled Transition Systems Analyser (LTSA) tool to model the LTS as a set of interacting finite state machines. The properties required of the system are also modelled as state machines, and the LTSA performs compositional reachability analysis to exhaustively search for violations of the desired properties. The following subsection gives a formal definition of the LTSA and its usage.

2.1.1 Analysis of LTS

For the purpose of this project, we analyse LTSs according to fluents and events, as described in detail in Chapter 3. It is therefore crucial to familiarise ourselves with *Fluent Linear Temporal Logic* (FLTL) and any other related notions. This subsection provides the necessary descriptions, together with an introduction to the *Labelled Transition System Analyser* (LTSA).

FLTL is a *Linear Temporal Logic* (LTL) formalism consisting of time modalities which are used to model event-based systems. Given a set of atomic propositions P, a formula in LTL can be defined using the Boolean connectives $\wedge$, $\vee$, $\neg$, and $\leftarrow$, and the temporal operators $\bigcirc$ (*next*), $\Box$ (*always*), $\Diamond$ (*eventually*), $\bigcup$ (*strong*) and W(*weak until*).

Fluents are state predicates whose values are determined by the occurrences of initiating and terminating events, that make the fluent values evaluate to true or false, respectively. They are used in the stream of *Artificial Intelligence* [32] in order to reason about the effects that different events can have on the state of a system. For our purposes, we will consider only propositional fluents, defined by Miller and Shanahan [25] as the following:

Definition 1. *"Fluents (time-varying properties of the world) are true at particular time-points if they have been initiated by an action occurrence at some earlier time-point, and not terminated by another action 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."*

The informal definition comes from the *Event Calculus* (EC) that was originally introduced by Kowalski and Sergot [17] as a logic program framework for representing and reasoning about events and their effects.

In line with this definition, we define a fluent *Fl*, initially true or false at time zero, as a pair of sets where one represents the initiating events and the

other the terminating events. Note that the value of a fluent at the initial state is denoted as $Initially_{Fl}$, and Act denotes the set of all possible events. A fluent definition therefore takes the form:

$$Fl \equiv \langle I_{Fl},\ T_{Fl} \rangle \text{ where } I_{Fl},\ T_{Fl} \subset Act \text{ and } I_{Fl} \cap T_{Fl} = \emptyset$$

For the example LTS in Figure 1.3, we assume the fluents *PumpOn*, *HighWater*, and *Methane*, defined as

- $PumpOn \equiv \langle\ turnPumpOn,\ turnPumpOff\ \rangle$
- $HighWater \equiv \langle\ signalHighWater,\ signalLowWater\ \rangle$
- $Methane \equiv \langle\ signalCriticalMethane,\ signalNotCriticalMethane\ \rangle$

A fluent will hold at a state s if it holds at the initial state and continues holding at every state in the path between the initial state and state s, or if some initiating event has taken place, but no terminating event has occurred which would modify the fluent value at the state being taken into account. A key point to note is that events immediately affect the values of fluents, and because the sets of initiating and terminating events for a fluent have to be disjoint, the value of a fluent is always deterministic with respect to a system execution. Fluents are completely defined by a fixed subset of well defined events, which enables more than one fluent to hold at the same state. We assume that there is a one-to-one mapping between an *action* in an LTS and an *event* in FLTL, so these terms may be used interchangeably.

LTSs can be seen as models corresponding to a given set of FLTL formulae, where system states are defined by the fluent values at each state. Similarly, De Nicola and Vaandrager introduced Doubly Labelled Transition Systems (DTSs) [28], which have actions labelling transitions, whilst propositions label the states. In a similar way, we use fluents for implicit state labelling.

The LTSA tool, developed by Jeff Magee, Jeff Kramer, Robert Chatley and Sebastian Uchitel (all from Imperial College London), is a very useful mechanism that enables different types of automated analysis over LTSs, including fluent model checking. It is not only a verification tool for concurrent systems, but can be considered as a general purpose tool for exploring and analysing event-based system specifications. It checks, mechanically, whether the specification of a concurrent system satisfies the required behaviour properties. In addition, it provides an animated version of the specification, hence enabling a more interactive analysis of system behaviour.

Since the main objective of this project is to work on existing LTSs and ensure that these can be refined to only accept positive system behaviours according to a set of goals and conditions, whilst rejecting all unwanted or negative ones, this report will include extensive references to them and will exemplify them using LTSA-generated model diagrams. It is therefore highly recommended to be familiar with the basic concepts found in [23].

2.2 Scenario-Based Specifications

A *scenario*, just as it is referred to in common language, is a concrete description of an action or event, or a sequence of these. The events can either occur in the environment or be performed by the system itself. For example, in the case of the mine pump system mentioned previously, an event that occurs in the environment surrounding the system is the increase in methane or water levels, whilst an event performed by the system itself is the switching on/off of the pump.

Scenarios are widely used in the pre-requirements stage since it is easier for customers and domain experts to use than an abstract model. For our purposes, scenarios represent examples of desirable and undesirable behaviour of a system. As expected, desirable behaviour is represented by positive scenarios, which are assumed to be consistent with the system specification, whilst undesirable behaviour is represented by negative ones (inconsistent with system goals).

More formally, a scenario can be defined [2] as a finite trace $\langle e_1, ..., e_n \rangle$ that describes a system's hypothetical behaviour from its initial state by specifying the events that are initiated by the system in response to events taking place in the environment. A *positive* scenario $\langle e_1, ..., e_n \rangle^+$ implies that there should be at least one trace in the LTS representing the system specification where $\langle e_1, ..., e_n \rangle$ can be accepted as an input. Conversely, a negative scenario $\langle e_1, ..., e_n \rangle^-$ implies that there should be no traces in the LTS where $\langle e_1, ..., e_n \rangle$ is accepted as an input.

Scenarios have been represented in a myriad of different ways, including use case maps, sequence diagrams, and message sequence charts. We have chosen to use the latter because of the simplicity and ease with which they can be understood. A *Message Sequence Chart* (MSC) is used to describe and specify the interaction between different system components, users, and the environment through the combined use of a graphical and textual language. Vertical lines represent the timelines for each component, whilst horizontal arrows between components labelled with an event are used to

define the interaction between those components. As for the ordering of events, an MSC timeline defines a total ordering on the events that are incoming or outgoing from the component to which the timeline corresponds, whilst the entire MSC defines a partial ordering on all events. The LTSA is one of the tools which supports the drawing and subsequent automated verification of message sequence charts.

Given the LTS model in Figure 1.3, a positive scenario could be one where the water in the mine reaches a high level, which makes the pump turn on to control the water level before it overflows. Once a low water level has been reached, the pump can turn itself off. This is represented in the MSC in Figure 2.6. Conversely, we could also have a situation where, following the increase in water level, an action to turn the pump off is attempted. Turning the pump on in this case would have been more appropriate to control water levels, whilst the *turnPumpOff* action is undesirable, and hence constitutes a negative scenario, illustrated in Figure 2.7.

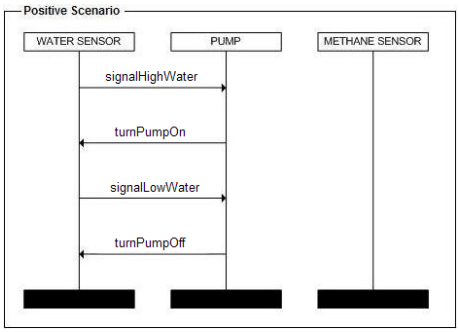

Figure 2.6: A positive scenario for the Mine Pump

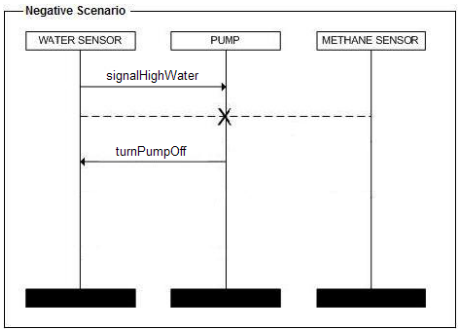

Figure 2.7: A negative scenario for the Mine Pump

Note that in the case of the negative scenario, a number of differences can be observed when comparing its MSC to that of the positive scenario. For example, the last message is crossed out, and is referred to as the *proscribed message*. This message is separated from previous messages by a dashed line, where the part above the line is called the *precondition*. This means that once a trace described by the precondition has occurred, the next event cannot be that defined by the proscribed message. The dotted line implies that *all* the events in the precondition must occur first, before the proscribed message.

The next chapter describes other approaches that we have studied in order to place our work into context.

Chapter 3

Related Work

A number of different attempts have been made within the Requirements Engineering field in pursuit of similar goals to our own, namely the discovery of behavioural models using scenarios. This discipline, which is also the first stage of the systems engineering and software development processes, is concerned with the analysis of high-level stakeholder goals so that they can be refined into system models such as LTSs which can be devised from formal requirements. Our project is aimed at the refinement of these system models through the use of scenarios.

The Requirements Engineering process can be very error-prone and may not provide the level of accuracy needed, so ideally an automated tool would be of great help. However, the various efforts have tried to generate behaviour models from scenarios, each having its benefits and drawbacks.

One method of synthesis is through the use of Message Sequence Charts representing the desired system behaviour in terms of interactions between system components, users, and the environment. This was developed by Uchitel et al. [34]. However, the graphical representations provided by this technique are often used for documentation purposes and for communication between the developers and the clients. Due to their limited use in the design phase of the software development process, Kruger et al. [19] proposed the translation of MSCs into state charts, to maintain their role in the requirements capture phase and in addition to enable their exploitation in later stages, such as refinement, validation and verification. Another technique involves having sequence diagrams capturing positive scenarios, as they can be understood by customers, software developers and requirements engineers alike, and UML state charts can then be generated from these [36]. A more interactive version of the latter method was developed

by Mäkinen and Systä [24], which poses acceptability questions to the user in order to avoid generating undesirable scenarios.

Other examples include the inductive learning technique suggested by Van Lamsweerde and Willemet [35], aimed at generating goal specifications in LTL from scenarios, or certain attempts to produce *Specification and Description Language* (SDL) specifications from MSCs (an example is specified in [14] for instance).

A slightly different approach to the previous ones is the L* learning algorithm presented in [6], which iteratively generates and refines assumptions based on queries and counterexamples, in order to determine whether a particular safety property holds in a system. The assumptions and properties are represented as safety LTSs, so the technique can be used to learn a regular language expressed through fluents and events, and is guaranteed to produce the minimal LTS that accepts the given language. We can relate to this approach because our algorithm is also trying to ensure that the refined system model adheres to the safety property(ies) that is(are) implicitly expressed by the given sets of positive and negative scenarios.

Our work relies on the existing achievements described in [2, 3, 8, 9, 10], which are explained in more detail later in this section.

3.1 Generalisation of LTS models from Scenarios

Similarly to the approach we present in this report, the synthesis process described in [8] considers *both* positive and negative end-user scenarios, which are incomplete, so that additional ones are generated. As further scenarios become available, the given LTS model is incrementally refined. The interactive element of the algorithm is an extension of RPNI [29], as it involves the users by asking them scenario questions.

Given the simplified LTS model of the Mine Pump in Figure 1.3, this methodology would consider some of the positive and negative scenarios for the system, and generate the *Prefix Tree Acceptor* (PTA) in Figure 3.1. PTAs represent each example taken from the system as a separate, unique path, where all paths start from a common unique initial state. The scenarios taken in this case are

- $\langle$*signalHighWater, turnPumpOn, signalLowWater, turnPumpOff*$\rangle^+$
- $\langle$*signalHighWater, signalLowWater*$\rangle^+$
- $\langle$*signalHighWater, turnPumpOff*$\rangle^-$

Note that due to space constraints we have replaced the *signalHighWater* and *signalLowWater* events with *highWater* and *lowWater*, respectively, and similarly with the *turnPumpOff* and *turnPumpOn* events, which have been replaced with the labels *turnOff* and *turnOn* in the PTAs. However, the *highWater* label is different from the *HighWater* fluent label, and in fact constitutes an initiating event for this fluent.

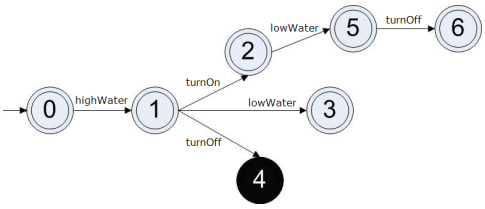

Figure 3.1: A PTA for the simple mine pump (a)

A state-merging algorithm is then proposed for the initial PTA. A path leading to a black state in the PTA is one that is *rejected* by the LTS, whilst one ending in a *grey* state is *accepted* by the LTS. It is therefore never the case where a pair consisting of a grey state and a black state is considered for merging. states 0, 1, and 2 are referred to as *consolidated* because they cannot be merged according to the *compatibility* of states defined in [8]. The first pair of mergeable states considered is the one including states 0 and 3, which computes a quotient automaton to generalize the current set of accepted behaviours. As a result of the merging, the new combined state is labelled as the state in the pair which has the lowest rank (in this case state 0). The resulting LTS is shown in Figure 3.2.

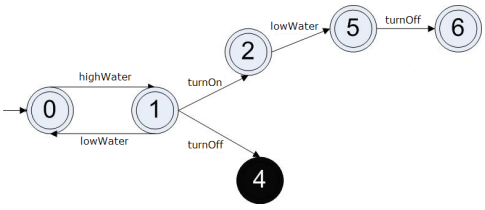

Figure 3.2: A PTA for the simple mine pump (b)

After some further merging of states, which we do not describe in detail here because of space constrictions, the LTS in Figure 3.3 is returned. Details regarding the actual merge can be found in [8].

[8] presents tool-supported techniques that overcome two of the problems that are commonly faced by recent efforts made in the area of synthesising

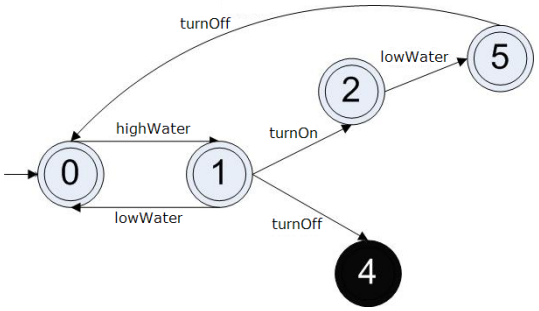

Figure 3.3: A PTA for the simple mine pump (c)

requirements or behavioural models inductively from scenarios. These are, namely, the need to add additional input other than scenarios to the methodology, such as state assertions, which makes it difficult for non-technical users to use it, and the difficulty in understanding the generated state machines because of the lack of domain-specific properties. The issues are addressed by extending known learning techniques for grammar induction, described extensively in [8]. The general idea is to successively merge state pairs from the initial LTS, and at each stage the LTS covers all positive scenarios and excludes all the negative ones. *Merging* two states implies representing them as a single state, after having checked that the resulting LTS would correctly reject all the negative scenarios. This helps generalise the behaviour that is currently accepted by the system.

An automated process is introduced in [9] for the approach in [8], based on the use of knowledge about the target system, which can help constrain the induction process, and hence ask the end-users less scenario questions. Apart from this, it helps to produce a more adequate LTS model, which is consistent with the additional information. This extra knowledge includes fluent definitions, domain properties, and system goals.

Even though the output of this automated process is the same as the one presented in [8], the actual merging of states is different. A "generate-and-test" algorithm performs an exhaustive search for equivalent state pairs, which are merged into equivalence classes. The equivalence of two states is determined according to the commonly known binary relation in mathematics, as described in [8, 9]. As a result, the merging is not performed in various steps as it was previously done in [8], but instead, the set of equivalence classes denotes the states in the final LTS.

Applying this automation to the PTA in Figure 3.1 would result in the following partition into equivalence classes, with the final LTS illustrated in

Figure 3.4 - note that the result is the same as the one obtained previously, but this time there are no intermediate LTSs generated, since the equivalence classes are defined altogether and the result is generated based on these classes.

$$\pi = \{\{0, 3, 6\}, \{1\}, \{2\}, \{4\}, \{5\}\}$$

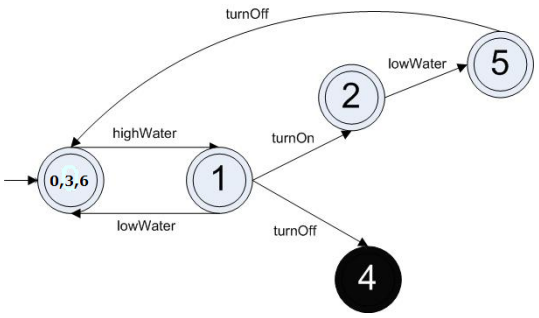

Figure 3.4: Resulting PTA for the simple mine pump (using automation)

3.2 Using ILP to Extract Operational Requirements from Scenarios

One of the approaches that we have studied in the context of generating behavioural models from scenarios is via the use of ILP. This constitutes one of the families of *inductive learning* methods, whose aim is to obtain general domain knowledge from the specific knowledge that is provided by domain examples. Inductive learning is most commonly referred to as *learning by example*, where a system *induces* a general rule from a set of observed instances. In the case of ILP, examples and domain knowledge are represented by Horn clauses.

The learning task presented in [2, 3] also uses scenarios. Stakeholders usually convey system properties through a more intuitive and narrative style, by using scenarios that provide only a partial description of a system in terms of its desirable and undesirable behaviour. In practice, a high level of time-consumption and inaccuracy can be experienced by extracting formal requirements from these kinds of specifications. The ILP method helps overcome these problems by using a partial system specification which can be extended with event pre-conditions and trigger-conditions from the

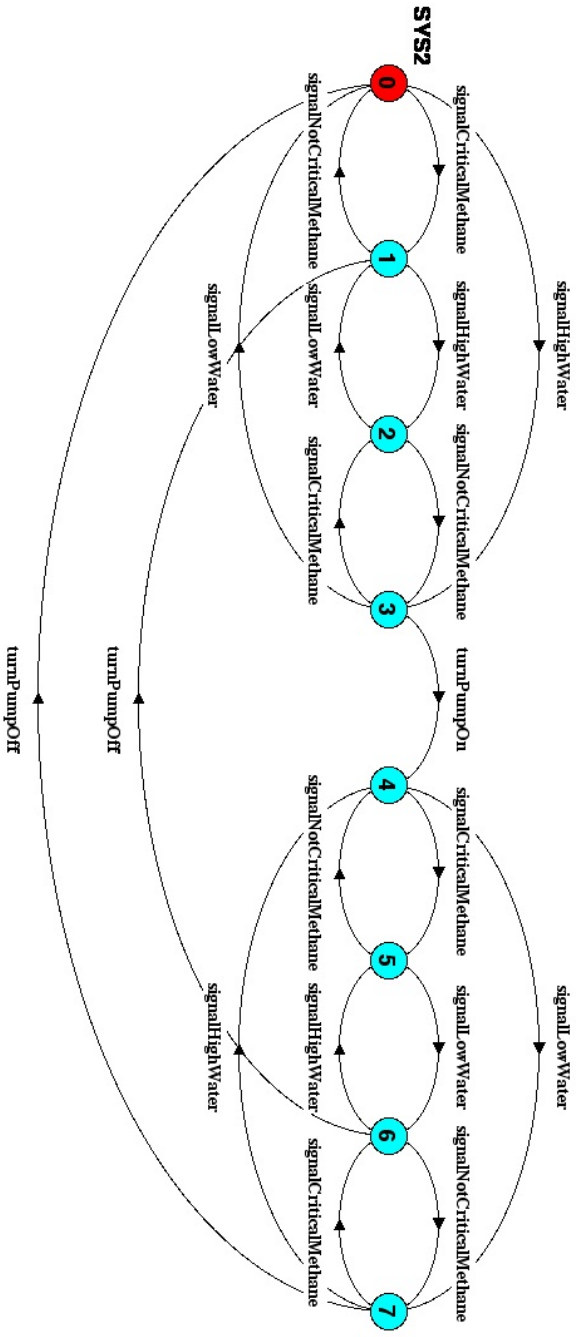

Figure 3.5: Results of ILP approach

information contained in user-provided scenarios. It therefore tackles the problem through a declarative representation of behaviour models. The application of this technique to the simple LTS model shown in Figure 1.3 would give the following solution. A diagrammatic representation has been included to help visualise the result in Figure 3.5.

$$\Box(\mathit{HighWater} \longrightarrow \bigcirc\ \mathit{!turnPumpOff})$$

This solution constitutes a pre-condition, and in plain English it means: "It should not be possible to execute a *turnPumpOff* event at a state where the fluent *HighWater* is true".

By taking advantage of the semantic relationship present between LTL and *Event Calculus* (EC), the initial specification represented in LTL is transformed into an EC logic program that becomes the input to the ILP system, which in turn learns the missing requirements of the system in question. A non-monotonic ILP system, called *eXtended Hybrid Abductive Inductive Learning* (XHAIL) [30, 31] is used to generalise the scenarios with respect to the *requirements specification*.

As formalised in [2], a requirements specification is comprised of a set of *initial state axioms* [11, 13] stating which fluents are initially true and which are false; *persistence axioms* [15, 17] formalising the law of inertia that any fluent will remain true (resp. false) until a terminating (resp. initiating) event occurs that causes it to switch state; *change axioms* [18, 20], stating that, for any fluent $f \in P_f$, the occurrence of any initiating (resp. terminating) event will cause f to become true (resp. false); and a set of *event precondition axioms* [21] which disallow any models that include transitions of the form $s_k \xrightarrow{e} s_{k+1}$ for any state s_k that satisfies a certain conjunction of fluent literals $\bigwedge_{0 \leq i \leq n}(\neg) f_i$.

In order to fully define the process of inductive learning, a translation of *scenario properties* is required, by expressing the initially incomplete specification and set of examples in LTL, and then transforming this to ILP through the use of the EC [17, 26].

A *scenario property* is an LTL formula corresponding to a scenario. When dealing with scenario properties, a positive scenario is said to have an *existential scenario property* because it is expected to hold in at least one path of a model, whilst the property associated with a negative scenario is expected to hold in all paths of a model and is therefore referred to as the *universal scenario property*.

The learning task pre-processes a system specification and given sets of positive and negative scenarios as presented in Chapter 2, in order to output a different FLTL which satisfies the required scenario properties, as formally defined in Definition 2.

The procedure of learning pre-conditions and trigger-conditions is stated in [3]. Given an initial requirements specification *Spec*, a set of undesirable scenarios *Und* and a set of desirable scenarios *Des*, the objective is to acquire the knowledge of a set of event precondition axioms *Pre* that will entail the negation of each undesirable scenario when added to the specification, and will also be consistent with the desirable scenarios. Definition 2 specifies two conditions that should theoretically be satisfied as a result: Firstly, that in any model of $Spec \cup Pre$, there is no path producing an undesirable scenario from *Und*; secondly, in any model of $Spec \cup Pre$, there is always a path that corresponds to each desirable scenario in *Des*. If both these conditions are fully satisfied by a set of event precondition axioms, then the set is described as a *correct extension* of the requirements specification with respect to the given scenarios, and this determines the completion of the process.

Definition 2. *Let Spec be a requirements specification, Des be a set of desirable scenarios, and Und be a set of undesirable scenarios. A set Pre of event precondition axioms is a correct extension of Spec with respect to Des and Und iff*

- $Spec \cup Pre \models_M \neg P_u$, *for each undesirable scenario* $P_u \in Und$
- $Spec \cup Pre \not\models_M \neg P_d$, *for each desirable scenario* $P_d \in Des$

The final stage of the learning process involves taking the LTL specifications and scenarios mentioned above, and translating them into EC normal logic programs. This methodology is explained in a detailed manner in [2, 3], where the necessary background material on the Event Calculus is provided as well. It is assumed that pre-condition axioms are to be learnt for the *last* event of each universal and existential scenario property, because that particular event will determine whether the path till then can be accepted by a model or not, with respect to the desirable and undesirable scenarios. Hence, each universal scenario property produces a sequence of facts which state what events certainly happen, following a fact stating that some event should not occur immediately afterwards. Each existential scenario property states that a certain sequence of events should happen.

3.3 Refinements of LTS models based on partition-refining

A well-known learning method which starts off with the most specific system specification, and progresses to the most general form, is state-merging in an FSA (*Finite State Automaton*), as referred to in multiple sources including [4]. Different algorithms of this type are reviewed in [27], but their outcome is a *generalisation* of the information found in the initial automaton. However, we are more interested in a *specialisation* approach, which can lead to a refinement of the LTS corresponding to the analysed system. A similar technique, called the *master algorithm*, is described in [10] to learn subclasses of regular languages by searching a partition over the states of the initial automaton, which we refer to in this subsection.

Different criteria can be used for the partition refinement task, and so different instantiations of the master algorithm are possible, but the results obtained both with the *k-equivalence* and the *k-reversibility* criteria are the same in our example. Similarly to [8, 9], the first stage of Elomaa's approach also involves the construction of a PTA, but this time it is only from the given *positive* examples, followed by the iterative refinement of states into several equivalence classes. The implementation starts off with a single block containing all the states (as opposed to beginning with all the states forming their own partition blocks for state-merging), which is then decomposed into multiple blocks until all the blocks in the partition are consistent with the properties defined for the system. We can apply this partition-refining technique to the initial PTA specified in Figure 3.1, but without taking into account state 4 and its incoming transition, as this constitutes a negative scenario, which the partition-refinement does not take as input. Figure 3.6 illustrates the initial PTA corresponding to the positive scenarios

- $\langle signalHighWater,\ turnPumpOn,\ signalLowWater,\ turnPumpOff \rangle^+$
- $\langle signalHighWater,\ signalCriticalMethane,\ signalNotCriticalMethane,\ turnPumpOn,\ signalLowWater,\ turnPumpOff \rangle^+$

Once again due to space constraints, the events *highWater* and *lowWater* are replacing the *signalHighWater* and *signalLowWater* events, respectively, so they constitute the initiating and terminating events for the fluent *HighWater*. Similarly, the *signalCriticalMethane* and *signalNotCriticalMethane* labels have been simplified to read *methaneAppears* and *methaneLeaves*, respectively, for this example.

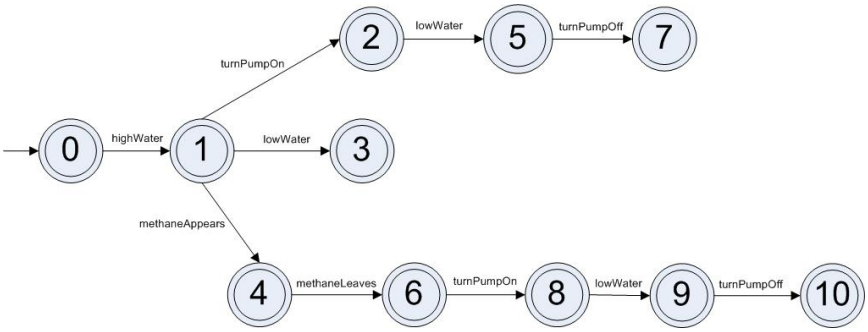

Figure 3.6: Initial PTA to Partition-Refinement

The algorithm usually starts off from a binary partitioning of states in which the final states make up one block, and the non-final states make up another. In our case, since we are operating on LTSs (in which all the states are final), then we consider all states to be part of the same block, which represents one 0-equivalence class (meaning that starting from any of the states, no traces of length at most 0 are accepted). The next stage would be to find corresponding 1-equivalence classes which can split up this single block of states into multiple ones. We observe that state 1 is not equal to any other in this case, because starting at state 1, the traces of length 1 that we can find are

- $(\boldsymbol{1} \xrightarrow{turnPumpOn} \boldsymbol{2})$
- $(\boldsymbol{1} \xrightarrow{lowWater} \boldsymbol{3})$
- $(\boldsymbol{1} \xrightarrow{methaneAppears} \boldsymbol{4})$
- ϵ

All these four traces of length 1 cannot be obtained by starting at any other state of the PTA, leaving state 1 in a partition block by itself. However, states 2 and 8 can be joined together into one block since both of them enable a trace of length 1 which is the same, namely the transition *lowWater*. Similarly, states 0, 3, 7, and 10 can execute exactly one trace of length 1 which is exactly the same, i.e. ϵ, hence causing these four states to form another block. This way, by using *k-equivalence* of states, the algorithm keeps constructing subsequent equivalence classes iteratively as long as there are blocks that need to be refined. As we know that each iteration bisects

one block, then the maximum number of refinement steps needed will be the number of states in the original PTA.

Eventually this would generate the equivalence classes

$$\{\{0,3,7,10\}, \{1\}, \{2,8\}, \{4\}, \{5,9\}, \{6\}\}$$

and the final output is illustrated in Figure 3.7. The algorithm actually returns the same equivalence classes as those obtained through the generalisation of LTS models described at the beginning of this chapter; the resulting PTAs from both methods are equivalent if we ignore the negative scenario in Figure 3.3 (since no negative scenarios were considered in Section 2.3) and states 4 and 6 in Figure 3.7 (because we did not consider the scenario $\langle highWater,\ methaneAppears,\ methaneLeaves,\ turnPumpOn,\ lowWater,\ turnPumpOff \rangle^+$ in the bottom-up generalisation approach in Section 2.1).

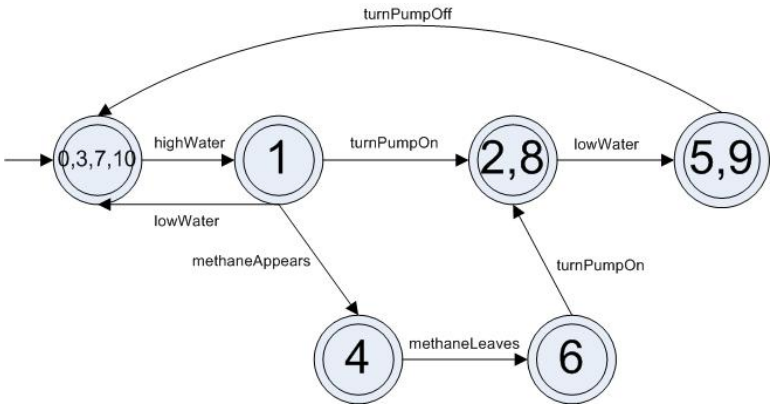

Figure 3.7: Final Output from Partition-Refinement

As stated in [10], the state minimisation algorithm for LTSs is the best partition-refining algorithm [12], and it also ensures the inference of the minimal, canonical automaton. [10] describes the approach through its application to various classes of well-known regular languages, such as the k-reversible languages [4].

In [36], independently-generated scenarios in the form of sequence diagrams are translated into state charts in order to eliminate the ambiguities and inconsistencies resulting from the amalgamation of these scenarios. Since the scenarios may portray the same, or very similar behavioural patterns, the algorithm defined in [36] adds semantic information to merge these scenarios whilst preserving the behaviour intended by the user. A separate

FSP is generated for each sequence diagram, and domain theory is then used to identify the same nodes from each FSP, resulting in the merging of these nodes. Users' input is required to classify the necessary generalisations and hence enable the algorithm to output a resulting state chart with fewer nodes.

The next chapter outlines in a detailed manner our approach for the refinement of LTSs.

Chapter 4

Algorithm for LTS Refinement

4.1 Introduction

As we described previously, the objective that we intend to achieve with this project is:

1. To accept as input an initial system model (an LTS), a set of positive scenarios, and a set of negative scenarios, where both sets are accepted by the initial LTS.

2. To use the input scenarios in order to refine the system model, so that an LTS model is generated as output, which still covers all the positive traces but simultaneously rejects the negative ones.

The process is illustrated in Figure 4.1.

Let us consider our running example of a Simple Mine Pump Control system. We have included the corresponding LTS in Figure 4.2. We intend our algorithm to be able to work on such an LTS and deliver a refined LTS such as the one returned by the ILP approach, as displayed in Figure 4.3.

This chapter outlines the algorithmic details related to the refinement.

4.2 Representation of Scenarios as PTAs

The very first step in the refinement algorithm is to construct PTAs corresponding to the give positive and negative scenarios. Each scenario is

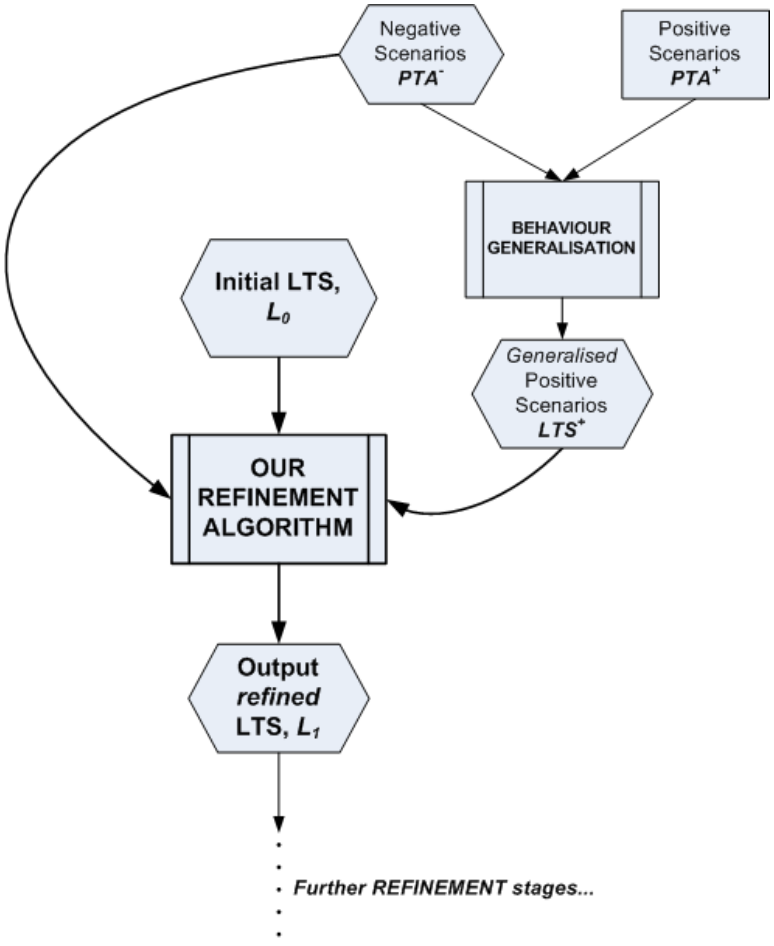

Figure 4.1: Our algorithm structure (a single refinement step)

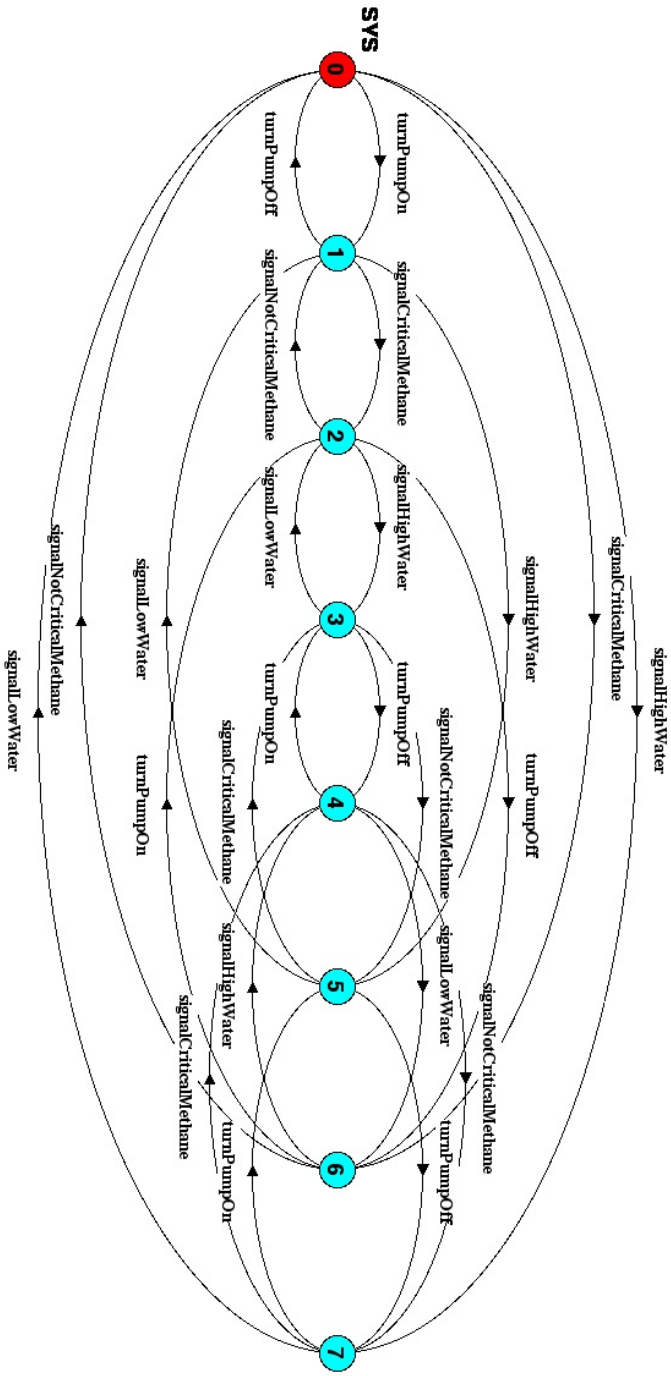

Figure 4.2: An example LTS for a Simple Mine Pump

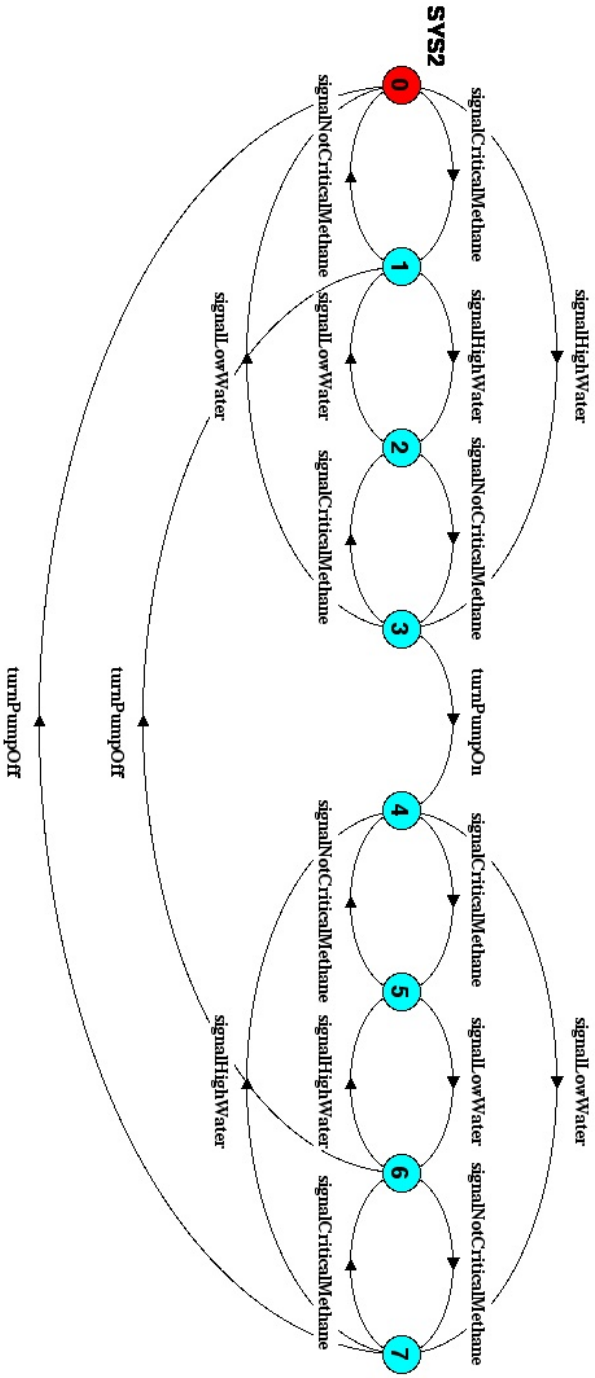

Figure 4.3: What we intend to achieve

represented on a separate branch in the PTA, so that we can use information regarding individual states and transitions during later stages of the refinement process. Scenarios are assumed to start in the unique initial state of the PTA, which is defined by the same fluent values as those in the initial state of the given LTS model. The pseudocode can be viewed in Algorithm 1.

Algorithm 1 Pseudocode for PTA Construction

```
1: INPUT: List of scenarios; List of fluents in initial state of LTS
2: OUTPUT: Tree representation of scenarios, pta

3: create tree with root node state 0, defined by the same fluents as fluents
4: set the index of the next state, stateNo = 1
5: for (each scenario s in scenarios) do
6:    parent = root
7:    for (each event label e in s) do
8:       create a node n with index stateNo
9:       add (stateNo,e) to the list of successors of parent
10:      add (index of parent,e) to list of predecessors of n
11:      add transition (parent,e,n) to list of transitions of pta
12:      parent = n
13:      stateNo = stateNo + 1
14:   end for
15: end for
```

Accepting States It is a vital point to note that in the PTA corresponding to positive scenarios, *all* states are accepting states (denoted by concentric circles), since a *prefix* of any length of any of the scenarios also constitutes a positive scenario, and so there is no need for a trace to have to go through each of the states of a branch to be considered as an acceptable trace. Hence, the relevant code for constructing the positive PTA includes a line which sets *each* state to be *accepting*.

On the other hand, in the case of the PTA for negative scenarios, only the *final* state of each branch is considered as an accepting state, because we consider the *last* transition in a negative trace as the unwanted one, and which therefore leads to an undesirable state. This transition is therefore the most significant one compared to all other transitions in the same branch, as the former conveys useful information regarding undesirable system behaviour. Prefixes of a negative trace constitute acceptable system

behaviour, as they only include non-final states, which are *not* accepting in the case of the negative PTA. Hence, these prefixes would be *rejected* as traces in the negative PTA, since they do not convey *negative* behaviour. Thus, the initial and non-final states of the negative PTA need to be non-accepting. This is ensured by including the appropriate line of code after the inner for-loop in Algorithm 1, which sets the *last* state of every branch in the negative PTA to be *accepting*.

In addition, we need to emphasise that the significance of an *accepting* state in both these cases is not related to the fluent values holding at that state (i.e. the state's label), but is more to do with the actual events leading to that state. In the case of the positive PTA, each state is accepting because we assume that the system can undergo any event that is just a prefix or part of a positive trace, and it can end one of its executions there. As mentioned previously, the events that can occur starting at a state will vary depending on the fluents that hold at the particular state, but as long as they constitute part of the system's desirable behaviour, there is no requirement as to the number or type of events that need to take place in order for a trace to be considered as accepted. Similarly with the negative LTS, the sole reason for making only the *last* state of each path final, is not because of the fluents holding at those states, but is due to the unwanted incoming transitions into those states, which we want to capture and hence remove from the general LTS.

However, due to the nature of PTAs, it is often the case that the resulting models include more states and transitions than are needed, because of repetitions occurring across the different branches (for example, scenarios can have common transitions). It is therefore feasible to reduce these by merging states in order to speed up the overall algorithm.

The result of subsequent merges may be a system model with the minimum number of states possible, which maintains or *maximises* the positive behaviour. Note that a synthesised model may portray more positive behaviour as a consequence of merging states in the initial PTA, since additional traces may be entailed by the resulting model. For example, the presence of loops in the synthesised model would allow an indefinite number of certain transitions to occur, hence expanding the set of possible traces. Since the merging process does *not* eliminate any of the existing transitions, it is never the case that the synthesised model will entail *less* positive behaviour.

There will always be a reduction in the number of states of the initial PTA if at least one merge is performed. Nevertheless, the minimality condition is only satisfied when we obtain a synthesised positive LTS which

best represents the system's desirable behaviour, using the least number of states; in other words, it is not possible to obtain a different synthesised LTS with a smaller number of states, and which produces the same or an even greater number of positive traces. The *Discussion* subsection explains in a more detailed manner the reasons for which it is not always possible to obtain the minimal synthesised LTS.

4.3 Generalisation of positive scenarios

Having analysed related approaches for LTS generalisation in Chapter 2, we have identified various methods involving merging and/or partitioning of states, that could prove to be useful in our context. As a result, we were inspired by the partitioning of system states into quotient automata [8], and we used it to synthesise the LTS representing the given positive scenarios. The following subsections outline the steps involved.

4.3.1 Merging States

Let us first of all explain the meaning of *merging* states. Two or more states in a system are considered for merging if and only if they have the same fluent values; in simpler terms, the *same* fluents hold at each of those states. The result of the merge is a new single state whose incoming (resp. outgoing) transitions correspond to a union/disjunction of the incoming (resp. outgoing) transitions of the states that compose the merged state. As a result, the parents (resp. children) of the individual states become parents (resp. children) of the merged state. In addition, a merge between two states s and t, where s is the state with a lower rank than state t, will result in a merged state whose label is that of the state with the smallest rank in the merge. In this case, it is state s. This concept is formally defined in Definition 3, and an example is provided later in this section.

Definition 3. Consider an LTS M defined as the 4-tuple (Q,A,δ,q_0), where F denotes the set of fluents in the system. Let s and s' be two states in Q such that $\forall\, f \in F$, f is ***true*** at s *iff* f is ***true*** at s', and *rank(s)* < *rank(s')*. The result of merging states s and s' in M gives a new LTS M' defined by (Q',A',δ',q_0') *where*

- $Q' = Q - \{s'\}$
- $A' = A$

- δ' = $\delta \cup \{(s,a,t) \mid (s',a,t) \in \delta\} \cup \{(t,a,s) \mid (t,a,s') \in \delta\}$ - $\{(s',a,t) \mid (s',a,t) \in \delta\}$ - $\{(t,a,s') \mid (t,a,s') \in \delta\}$
- $q_0^{'} = q_0$

For a particular merge to be *valid* or *successful*, we need to ensure *consistency* between scenarios. For our purpose, the result of merging states in the positive PTA should be an LTS which does *not* include traces that entail any of the negative scenarios. Hence it is necessary to check the negative scenarios whilst constructing the generalised positive LTS to ensure this condition is satisfied. Let's exemplify this with an example to make the concepts clearer.

Let us consider a simple system where the set of possible actions *Act* = {*switchOn, switchOff*}, and the only fluent in this case is *On*, defined as *On* ≡<*switchOn, switchOff*>. If two positive scenarios for the system are $< switchOn, switchOff >$ and $< switchOff, switchOn, switchOff >$, then it can be represented by the simple 6-state PTA shown in Figure 4.4. On the other hand, a negative scenario for this system may be $< switchOn, switchOff, switchOn >$, if the system specification implies that the system can only be turned on once. The PTA for this is shown in Figure 4.5.

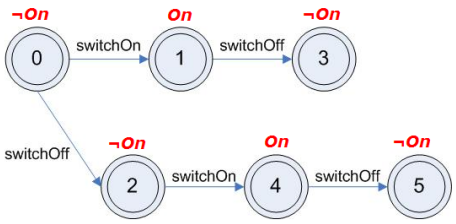

Figure 4.4: Positive PTA

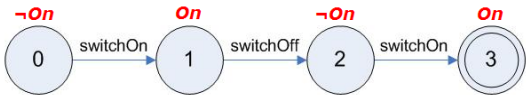

Figure 4.5: Negative PTA

If we were to perform state-merging on the positive PTA, we would consider states 1 and 4 for merging since both of these states are defined by the same fluent value, namely *On*, and states 0, 2, 3 and 5, because they are

defined by the fluent value $\neg On$. States 3 and 5 can be merged. However, state 0 cannot form part of the merged state, because doing so would enable an indefinite number of *switchOn* followed by *switchOff* events to occur, which would therefore disagree with our negative scenario (see Figure 4.6).

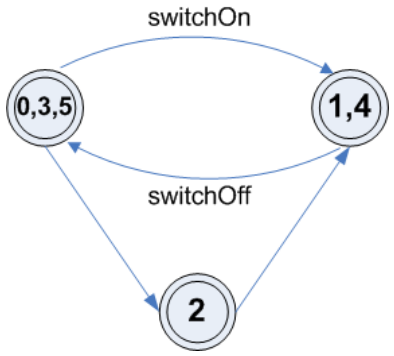

Figure 4.6: An incorrect merge for the positive PTA (a)

As regards to state 2, merging it with states 3 and 5 would also lead to an infinite number of *switchOn* followed by *switchOff* events, as shown in Figure 4.7.

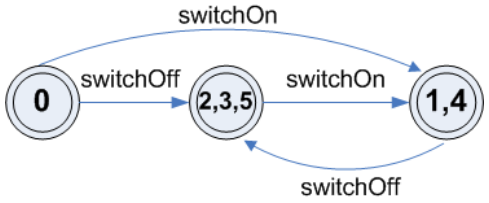

Figure 4.7: An incorrect merge for the positive PTA (b)

However states 0 and 2 can be merged together, and the correctly merged model is shown in Figure 4.8.

Figure 4.8: Correctly merged positive PTA

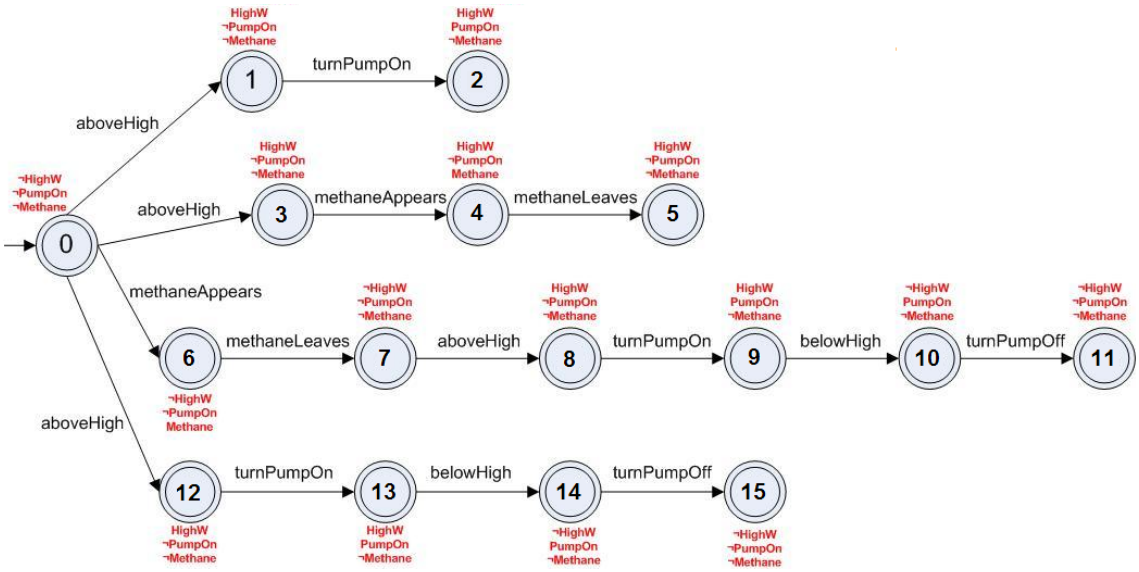

Figure 4.9: PTA for positive scenarios

4.3.2 State-Merging Algorithm

Following the definition of a *merge* in the previous sub-section, we can now outline the main points that need to be taken into consideration while merging states in the model shown in Figure 4.2 on Page 31.

The pseudocode for the procedure we use for coalescing states is included in Algorithm 2. The *TEST* procedure referred to in the state-merging code is shown in Algorithm 3.

Given the following positive scenarios,

- $\langle signalHighWater,\ turnPumpOn \rangle$
- $\langle signalHighWater,\ signalCriticalMethane,\ signalNotCriticalMethane \rangle$
- $\langle signalCriticalMethane, signalNotCriticalMethane,$
  $signalHighWater, turnPumpOn, signalLowWater, turnPumpOff \rangle$
- $\langle signalHighWater,\ turnPumpOn,\ signalLowWater,\ turnPumpOff \rangle$

and negative scenarios,

- $\langle turnPumpOn \rangle$

Algorithm 2 Pseudocode for State-Merging

1: *INPUT:* Positive PTA
2: *OUTPUT:* Synthesised Positive PTA *general*

3: a label is the string of fluents at a state, i.e. PumpOn & !Methane
4: *seenLabels* is the set of labels encountered so far in any of the states
5: set the generalised positive LTS *general* to be the same as input PTA
6: $groups = [(f, ls)|f \in seenLabels$ and ls is a list of lists of states]
7: **for all** state i in *general* **do**
8: $label$ = label of i
9: **if** ($label \notin seenLabels$) **then**
10: add *label* to *seenLabels* as it has been encountered
11: add (*label*,[[i]]) to *groups* as i is the only state with this label
12: **else**
13: extract element (*label*,[e|*list*]) from *groups* to get the indices of other states with the same label
14: add i to the end of list e in order to try merge i with states in e
15: $param = e$, as *param* is the set of states that i has been added to
16: **call** procedure **TEST**(*param*)
17: **if** (**TEST** returns false, i.e. merge not valid) **then**
18: **if** (*list* is empty, i.e. no other states to which i can be merged) **then**
19: create a new list e'
20: add i to e'
21: add (*label*,[e, e']) to *groups*
22: **else**
23: *there are more states to which i can be added*
24: list f = head of *list*
25: add i to f
26: $param = f$
27: ***go to*** *line 16*
28: **end if**
29: **else**
30: *the merge is valid*
31: state n = the lowest-rank state in *param*
32: **for all** (state s in *param*, s != n) **do**
33: set children states of s to be children of n instead
34: set parent states of s to be parents of n instead
35: **end for**
36: **end if**
37: **end if**
38: **end for**
39: **return** *general*

Algorithm 3 Pseudocode for TEST - Testing a merge

```
1: INPUT: list = list of states being merged
2: OUTPUT: true if the merge is valid, false otherwise

3: for all (state i in list) do
4:    for all (incoming trace p starting at root node and ending at state i)
      do
5:       for all (state j in list, i ≠ j) do
6:          for all (outgoing trace q starting at state j do
7:             if (the joint trace p + q covers a negative scenario) then
8:                invalid merge
9:                return false
10:            end if
11:         end for
12:      end for
13:   end for
14: end for
15: return true
```

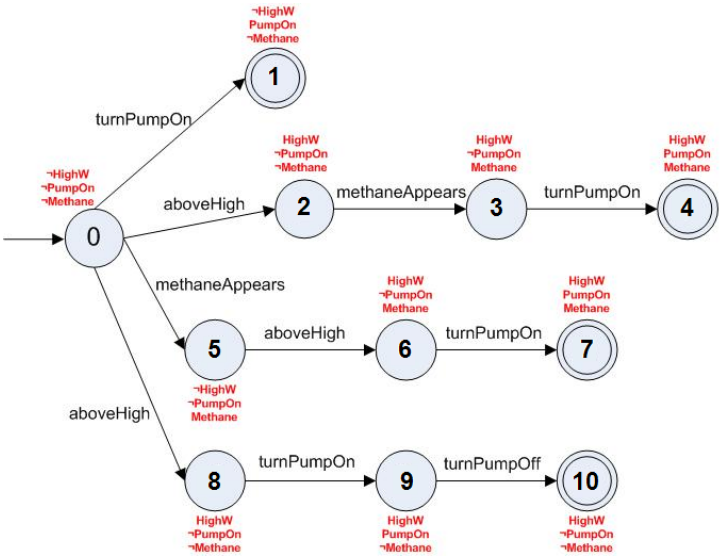

Figure 4.10: PTA for negative scenarios

- $\langle$*signalHighWater, signalCriticalMethane, turnPumpOn*$\rangle$
- $\langle$*signalHighWater, turnPumpOn, turnPumpOff*$\rangle$
- $\langle$*signalCriticalMethane, signalHighWater, turnPumpOn*$\rangle$

we can now devise smaller models representing these sets of scenarios using Algorithm 1, as shown in Figures 4.9 and 4.10. Note that we have replaced the *signalHighWater* and *signalLowWater* event labels by *aboveHigh* and *belowHigh*, respectively, to maintain a decent size for the PTAs. Similarly, event labels *signalCriticalMethane* and *signalNotCriticalMethane* have been replaced with labels *methaneAppears* and *methaneLeaves*. To avoid confusion, we will explain the various steps using the labels from the PTAs.

State-merging is performed on the PTA in Figure 4.9 as described in the previous section. We can intuitively see that there is room for at least one synthesising step - state 0 has more than one outgoing transition labelled with the action *aboveHigh*, which leads to multiple states having just the fluent *HighWater* holding. These could be merged (possibly with other states from the PTA) into one single transition, leading to one single state, and so on. Note that we do not necessarily need to take into account any particular order whilst carrying out the synthesis in this way, but for convenience, consistency, and ease of comprehension, our algorithm visits states using a depth-first search along each branch.

The algorithm starts off by looking at the initial state in Figure 4.9, i.e. state 0, which is defined by the fluent values $\langle\neg$*HighWater,* $\neg$*PumpOn,* $\neg$*Methane*$\rangle$. Since it is the first state visited, its fluent definition has not been encountered previously, and so the *if*-case at line 9 in Algorithm 2 is executed. However, states 7, 11, and 15 have the same fluent values as state 0. Thus, when they are visited, the *else*-case on line 12 is executed. Since states are visited in increasing rank order, states 0 and 7 are merged first, then states 11 and 15 are added to the same partition when they are visited (line 14). All merges are successful, hence the *else*-case on line 29 is executed for each merge. The resulting state is labelled 0, as this corresponds to the smallest rank (line 31).

Similarly, states 1, 3, 5, 8, and 12 are defined by the fluent values $\langle$*HighWater,* $\neg$*PumpOn,* $\neg$*Methane*$\rangle$, and can therefore be merged together into a new state numbered 1. This way, we continue merging states until no further merging between states is possible.

At the end of the merging phase, states can be renumbered in order to establish consecutive state numbers as we had before the states were merged.

For instance, if the result of merging states in a PTA results in an LTS with states {0, 1, 3, 5, 6, 9}, then some of the states can be renumbered so that we finally have an LTS with states {0, 1, 2, 3, 4, 5}. Note that this is not necessary, but is possible because the actual numbers in the states are just like state labels, and therefore insignificant with regards to the refinement process. In our example the resulting states would be {0, 1, 2, 3, 4, 6, 10}, which can be renumbered to {0, 1, 2, 3, 4, 5, 6}.

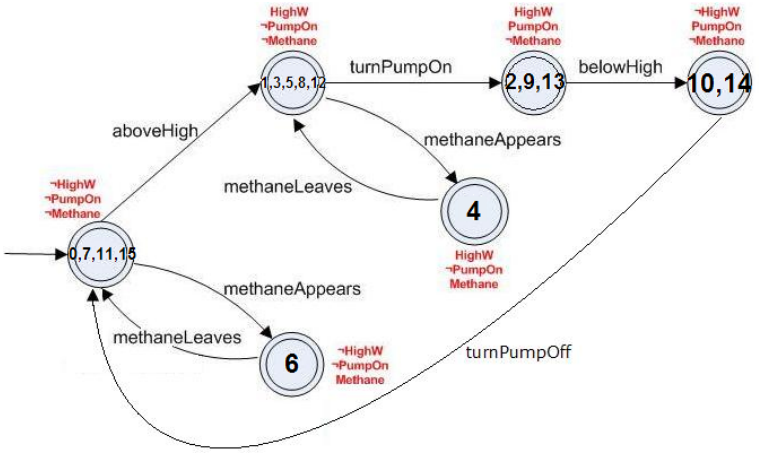

Figure 4.11: Synthesised LTS for positive scenarios

The final result of applying this state-merging algorithm to Figure 4.9 gives the LTS shown in Figure 4.11, which, if observed closely, portrays the same behaviour as the positive PTA, without having removed any of the previous possible traces, but with a smaller number of states. Additionally, we can see that the resulting model now has loops between state {0, 7, 11, 15} and state 6, as well as between state {1, 3, 5, 8, 12} and state 4, allowing an indefinite number of *methaneAppears* followed by *methaneLeaves* transitions to occur. Note that the result is actually an LTS, not a PTA, because common outgoing transitions from a single state are now grouped so that there is only one outgoing transition from that state with a particular action label, rather than having individual paths to denote each scenario. We remind ourselves that as a result of the generalisation, the number of positive scenarios covered by this LTS strictly includes those covered by the initial PTA given in Figure 4.9. In addition, we have performed a series of merges in this case, thus the number of states in the LTS is *strictly smaller* than those in the PTA.

4.3.3 Discussion

Minimality condition for State-Merging

We can define the smallest possible LTS corresponding to a PTA as being the one in which all possible merges from the initial PTA are successful. Thus, for every *unique* fluent combination that is found in the states composing the initial PTA, there is only *one* state in the smallest resulting LTS representing a particular fluent conjunction. In the case of the positive PTA in Figure 4.9, we can see that the different fluent conjunctions are:

- $\langle \neg HighWater \wedge \neg PumpOn \wedge \neg Methane \rangle$ - occurring in states 0, 7, 11, 15
- $\langle HighWater \wedge \neg PumpOn \wedge \neg Methane \rangle$ - occurring in states 1, 3, 5, 8, 12
- $\langle \neg HighWater \wedge \neg PumpOn \wedge Methane \rangle$ - occurring in state 6
- $\langle HighWater \wedge PumpOn \wedge \neg Methane \rangle$ - occurring in states 2, 9, 13
- $\langle HighWater \wedge \neg PumpOn \wedge Methane \rangle$ - occurring in state 4
- $\langle \neg HighWater \wedge PumpOn \wedge \neg Methane \rangle$ - occurring in states 10, 14

This means that the minimal LTS for this PTA would have just 6 states instead of the current 16, and in fact, luckily our merging algorithm returns the minimal LTS in this case, as shown in Figure 4.11.

Since our merging algorithm operates on the states of the initial PTA in increasing order of rank (denoted by the numbering of states), we studied an example to check whether this ordering notion makes a difference to the merging process. In order to do this, we chose to use a similar idea to the lattice of partitions defining quotient automata in [8]. Note that apart from the purpose of state-merging, the numbering of states plays no substantial role in the eventual refinement process; the numbers are merely there to establish a consistent order of traversal of the states.

Figure 4.12 shows a lattice of automata resulting from merges in an example of a positive PTA *without* taking into account negative scenarios, hence no matter what order the states are merged in, the final result is always the same (namely the automata with 6 states at the bottom of the lattice). However, the equivalent lattice that in addition takes into account negative scenarios, and therefore rejects particular merges of the states in the positive PTA if they cover any of the negative scenarios, is shown in

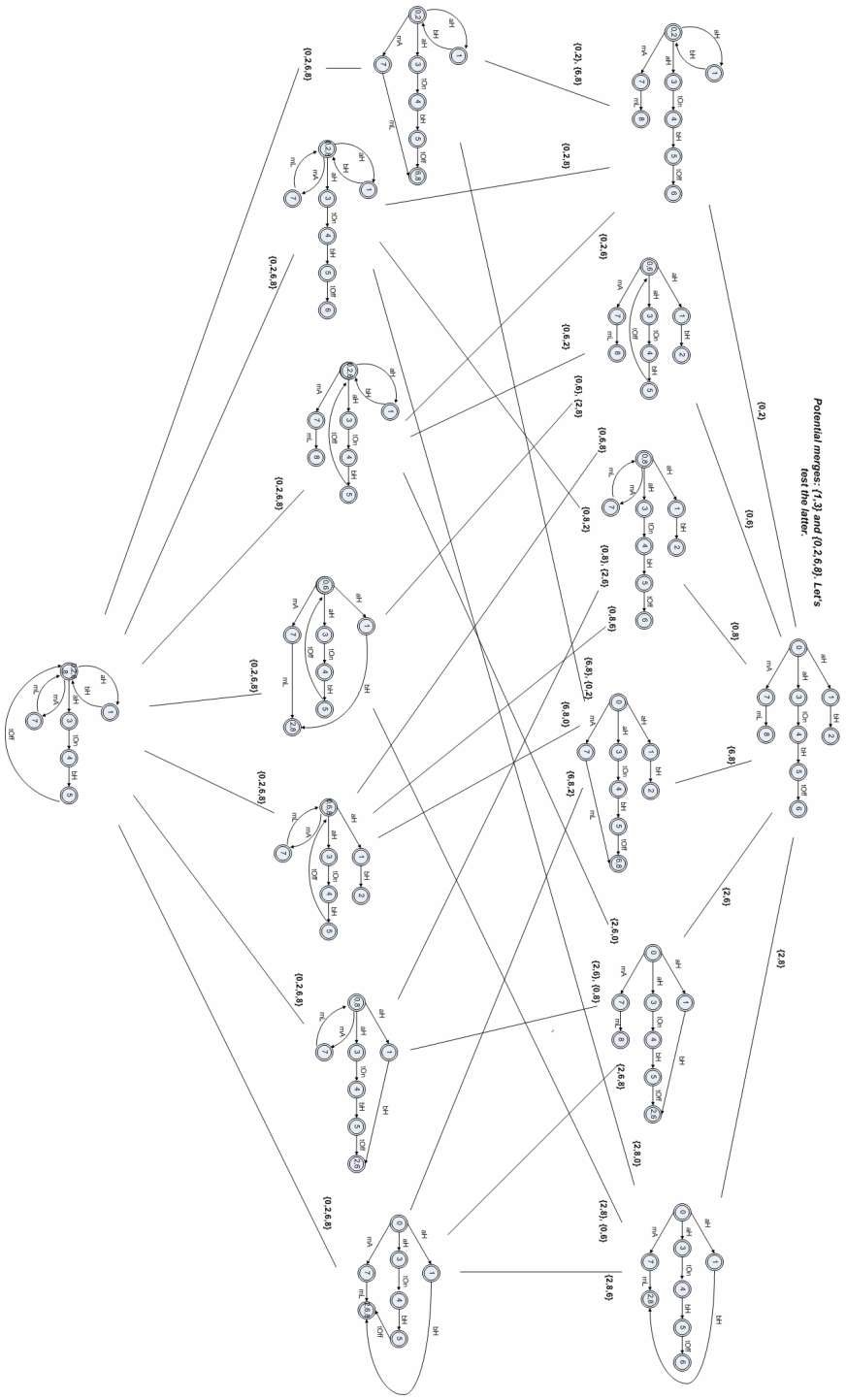

Figure 4.12: Lattice of possible state-merging steps

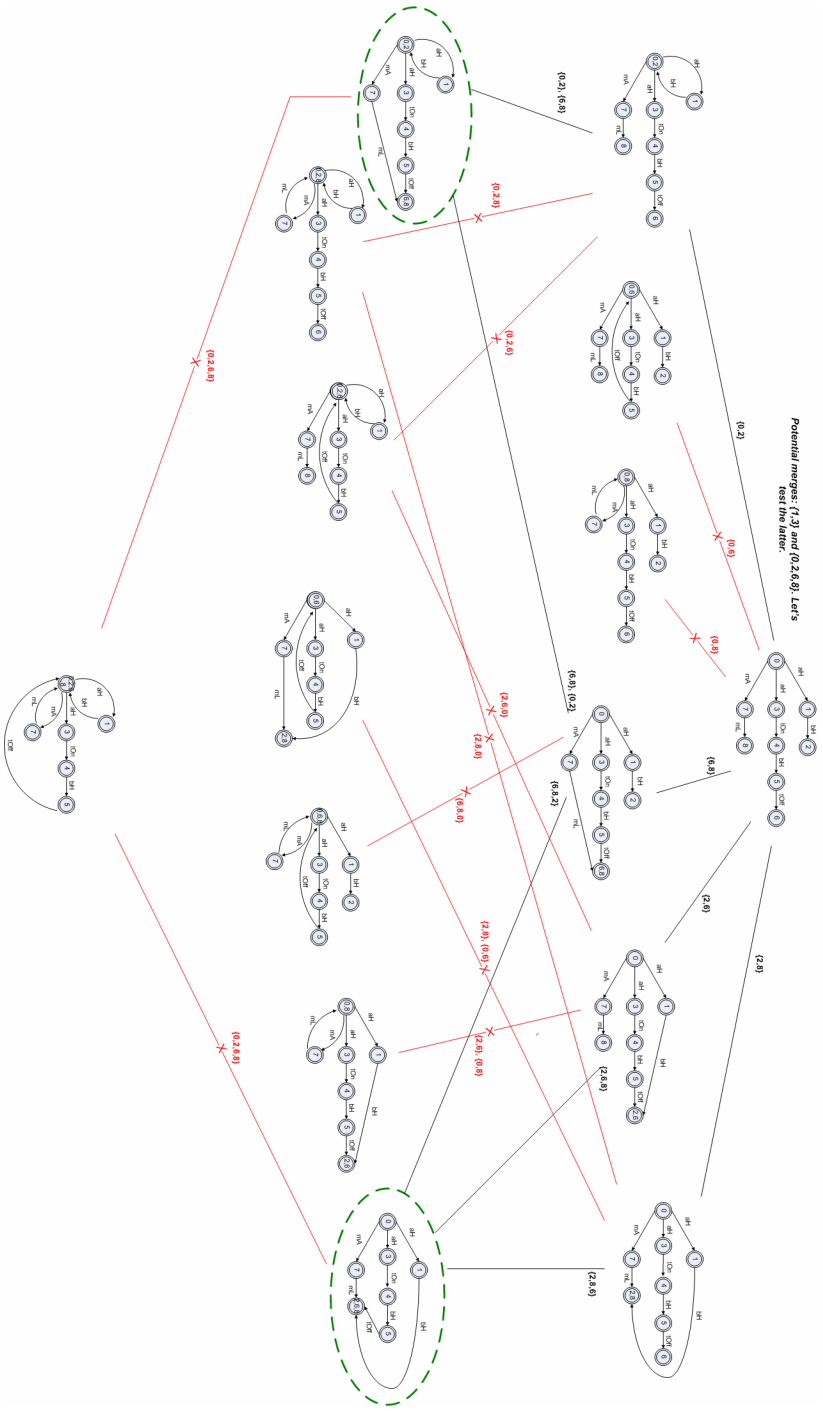

Figure 4.13: Lattice of possible state-merging steps (taking into account negative scenarios)

Figure 4.13. Here it is more evident that for a given set of positive scenarios in the PTA shown at the top of the lattice, the order in which states are considered for merging makes a difference to the final result, as each merge is greatly dependent on the previous merges, so the solution is not always unique.

Each lattice has multiple levels, and as we look downwards, the automata in each level (except the topmost one) correspond to more generalised system models, as they are a result of merging states from the previous level. Hence, the resulting models in one level have less states than the models in the previous level, but within the same level, all models have the same number of states. However, despite the generalisation of behaviour leading to a greater number of possible traces within the models of lower levels, the number of unit-length transitions stays the same throughout the lattice, as we are not adding any additional edges, and the merging does *not* get rid of any transitions that were present in the initial PTA.

Note that both the lattices illustrated are *fluent conjunction-specific*. In other words, here we have shown all the possible steps for a set of states which share a particular fluent conjunction, namely

$$\{\neg HighWater \wedge \neg PumpOn \wedge \neg Methane\},$$

but there may be another fluent conjunction that is shared by a set of *different* states in the system. It would be possible to perform state-merging using those states, hence further generalising the system. A similar lattice to the one shown in Figure 4.13 would need to be drawn for each fluent conjunction that leads to state-merging, to show the different paths that may be taken.

Our state-merging algorithm iterates over the first level of the lattice till it finds the first pair of states that will lead to a successful merge (this iteration corresponds to the *else*-case on line 10 of Algorithm 2, and the merge is validated at line 14). Thereafter, the method always tries to add states to the existing merged state (line 22), in the hope of eventually adding all potentially mergeable states for the same fluent conjunction into one single state, thus achieving the maximum reduction in the number of states as a result of that particular merge. It is not necessarily the case that the final LTS will have the least number of states, as explained in later sections of this report.

In this case the two different solutions for the this particular fluent conjunction are circled, and for ease of reference we have included them separately in Figures 4.14 and 4.15.

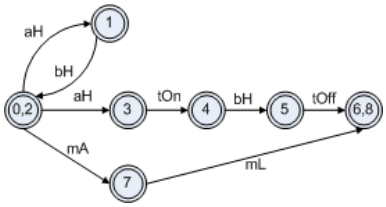

Figure 4.14: Solution 1

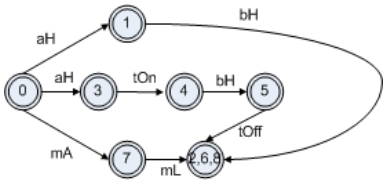

Figure 4.15: Solution 2

In this case, both solutions have seven states, and they have the same number of transitions as expected. The differing element, in fact, are the traces which are possible through each solution. The left-most one includes a loop with the event label *aboveHigh* followed by a transition labelled *belowHigh*. This means that an indefinite number of successive *aboveHigh*$\longrightarrow$*belowHigh* transitions would be considered as positive behaviour if we were to use this as our generalised positive LTS. The right-most solution in the lattice, on the other hand, does *not* include this and would therefore not allow this form of repetition.

Nevertheless, the LTS that is eventually generated and used as the generalised positive LTS will *not* influence the final results of our refinement algorithm. Despite the greater number of traces allowed by the left-most solution, our final algorithm does *not* look at entire traces while refining an initial LTS, but instead considers states and one-step transitions between them. As we mentioned earlier, all the different LTSs produced by the generalisation process for a given PTA include the same transitions, so no matter which solution we choose as our generalised positive model, the results of the overall refinement will be the same. The only difference lies in that each generalised solution *may* have a different number of states, and this could have an effect on the overall execution time of the refinement algorithm.

In fact, all the different minimal results of state-merging are valid, since they are all compared to the negative scenarios in order to ensure that we do not cover any undesirable behaviour with our resulting synthesised model, but we are interested in the one with the least number of states.

It may not always be the case that the different LTSs obtained have the same number of states, but we cannot guarantee that our algorithm will always choose the model with the least number of states and/or the maximum possible traces. In order to do this, we would need to go through each and every possible merge for each different fluent conjunction, and in each case we would need to accept the solution which has the least number

of states and/or maximum possible traces. This would imply storing each different automaton generated, so that they can all be compared eventually and the minimal one can be chosen. In real-life systems, this would involve a tremendous computational cost and would also increase execution time significantly.

Therefore, since all the different minimal generalised LTSs possible are correct, we allow for different solutions in our final refinement algorithm. All solutions are guaranteed to be correct given the generalised positive LTS and the negative PTA, and the only difference between one and another may be that one of them is more strictly refined than the other (i.e. one of them allows more traces than the other).

State-Merging in the Negative PTA

Following the same methodology used for state-merging nodes in the positive PTA, we could have also chosen to merge states in the negative PTA in Figure 4.10, such that the final outcome would *not* include any of the positive scenarios as acceptable traces, as this would incorrectly classify them as undesirable system behaviour, but it would accept all the negative behaviour included in the given negative PTA.

However, since we are mainly interested in the final states of the negative PTA and their incoming transitions as discussed previously, state-merging is not essential because all other states and transitions in the negative PTA are prefixes of some positive behaviour, and are therefore not used for pruning traces in the final refinement step. Hence for the purposes of our refinement algorithm, we are only merging states in the positive PTA.

4.4 Refinement Process

Now we cover the final, most important phase, namely the actual refinement of the system's LTS model. This works directly on the given initial LTS model for the system, without taking into account any declarative elements of the specification, and consequently outputs a new LTS which portrays all the desirable system behaviour, but none of the undesirable traces expressed by the negative scenarios.

We can use the synthesised LTS corresponding to the set of positive scenarios together with the negative PTA (that does *not* need to undergo state-merging as mentioned in the previous section) to prune unwanted paths from the general system LTS in Figure 4.17. There are two stages in this procedure, the first one being the initial phase involving the pruning of traces

that match transitions in the negative PTA exactly *(first-level pruning)*. The event labels corresponding to the pruned transitions are collected, and used in the next phase where further analysis is carried out to perform the refinement on the modified system LTS *(second-level pruning)*. However, we thought of two different ways of going about this. They both share the same initial step described in the next section, but thereafter they use different methods to deal with second-level pruning. The one that we decided to use is the *Fluent Conjunctions Method* which is explained later, whilst the alternative method referred to as the *Equivalent States Method* is included in the Discussion subsection.

4.4.1 First-Level Pruning

Algorithm 4 Pruning: Phase I

1: *INPUT:* neg = negative PTA, sys = initial LTS
2: *OUTPUT:* a refined LTS sys, which does not cover any of the negative traces but maintains all the positive ones

3: $victims$ = [(ev,st) | ev is the pruned event from state st]
4: $victims$ = []
5: **for** (each final state i in neg) **do**
6: event e = incoming event into i
7: $matches$ = list of states in sys with same fluents as i and incoming event e
8: **for** (each state m in $matches$) **do**
9: find a state n such that (n,e,m) is a transition in sys
10: add (e,n) to $victims$
11: remove transition (n,e,m) from sys
12: **end for**
13: **end for**
14: **return FURTHER_PRUNE**($victims$)

Let us consider again the model for the Simple Mine Pump Control system, illustrated in Figure 4.16. For simplicity reasons, we are going to work with an equivalent model illustrated in Figure 4.17, which only differs from the former in terms of some of fluent and event labels used. For instance, the fluent label *HighWater* has been replaced by *HighW*, so the set of fluents corresponding to this system is

{*HighW, PumpOn, Methane*}

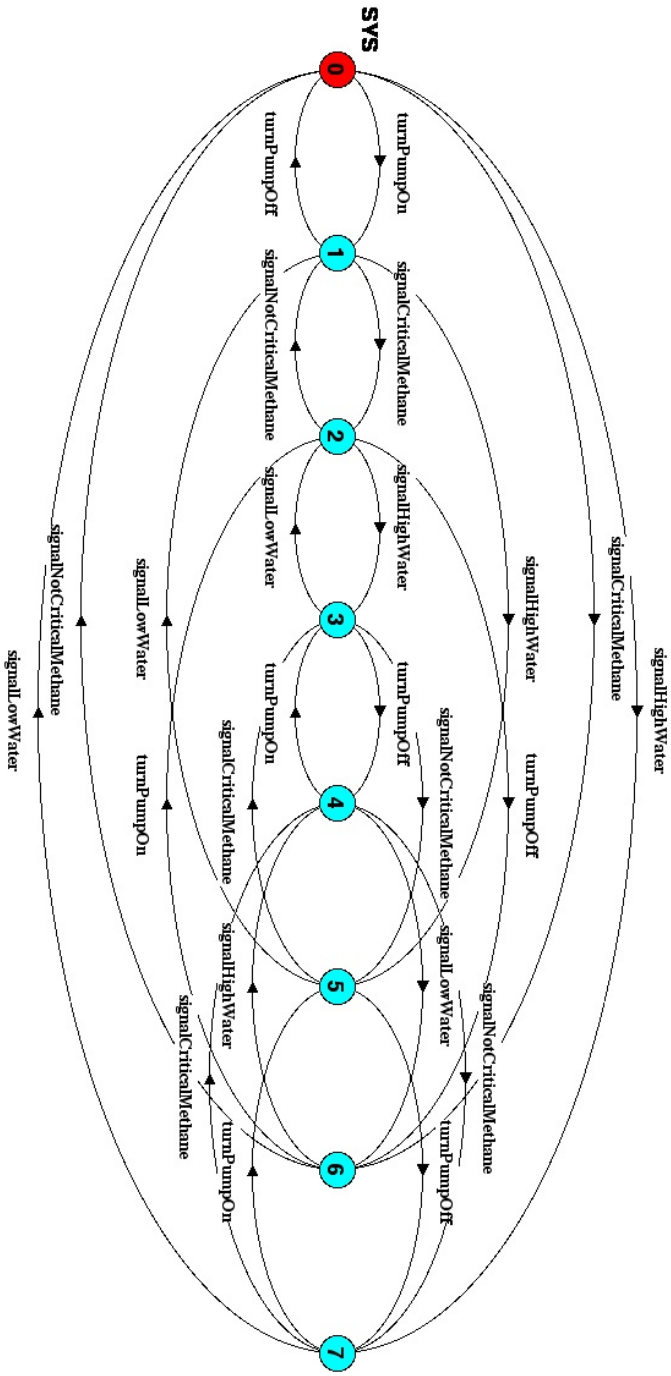

Figure 4.16: An example LTS for a Simple Mine Pump

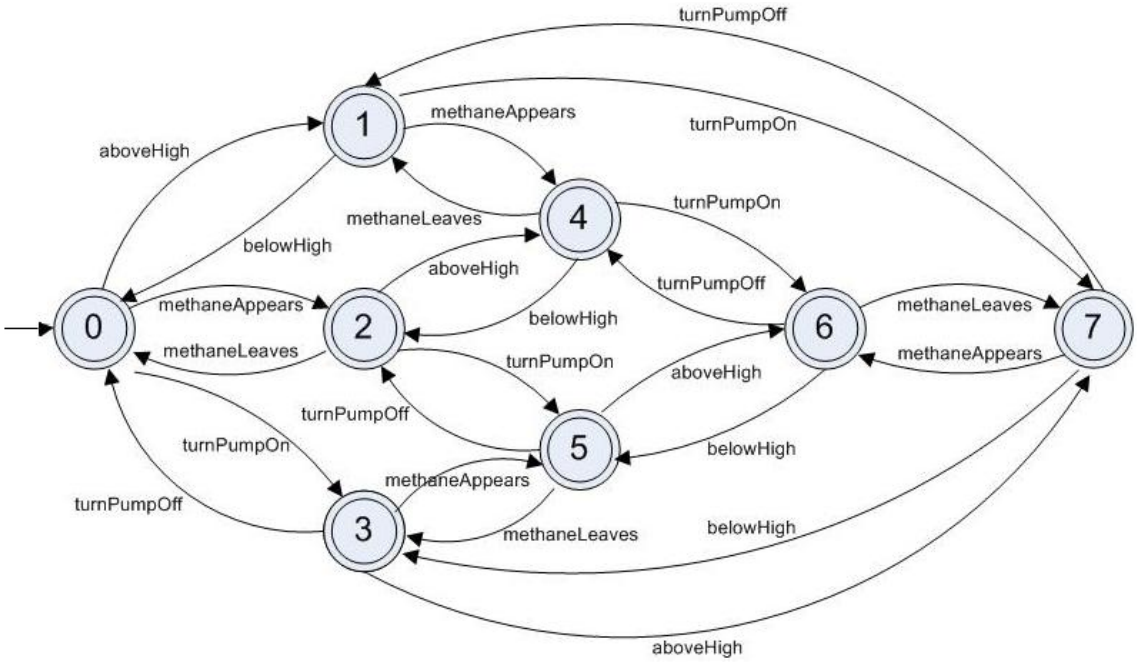

Figure 4.17: LTS for simple Mine Pump

The set of events is

$$\{aboveHigh,\ belowHigh,\ turnPumpOn,\ turnPumpOff,\ methaneAppears,\ methaneLeaves\},$$

which determine the following fluent definitions:

- $HighW \equiv \langle aboveHigh,\ belowHigh \rangle$
- $PumpOn \equiv \langle turnPumpOn,\ turnPumpOff \rangle$
- $Methane \equiv \langle methaneAppears,\ methaneLeaves \rangle$

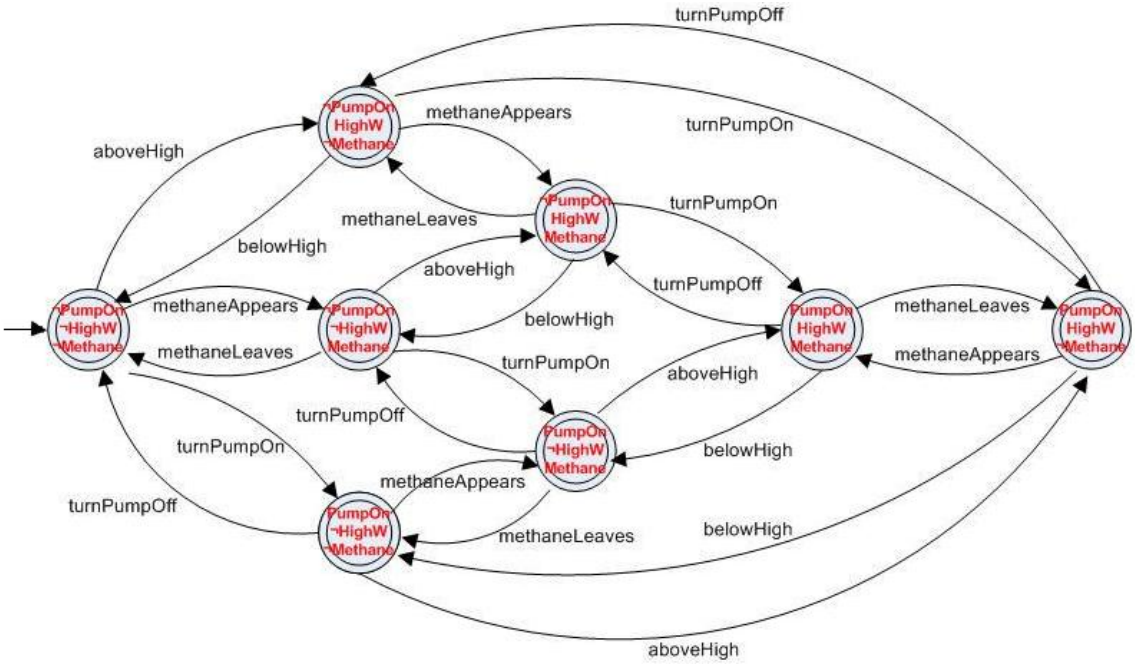

Figure 4.18: LTS for simple Mine Pump (with fluents)

The LTS diagram in Figure 4.17 is the most generalised model for the system, and hence contains the maximum number of states and transitions possible. To make the refinement of the initial LTS diagram easier, we can work with an LTS such as the one in Figure 4.18, where we can clearly see what fluents hold at each state.

Firstly, we use Algorithm 4 on this initial LTS. We find the states in the LTS which correspond to the final states of the negative PTA (i.e. those with

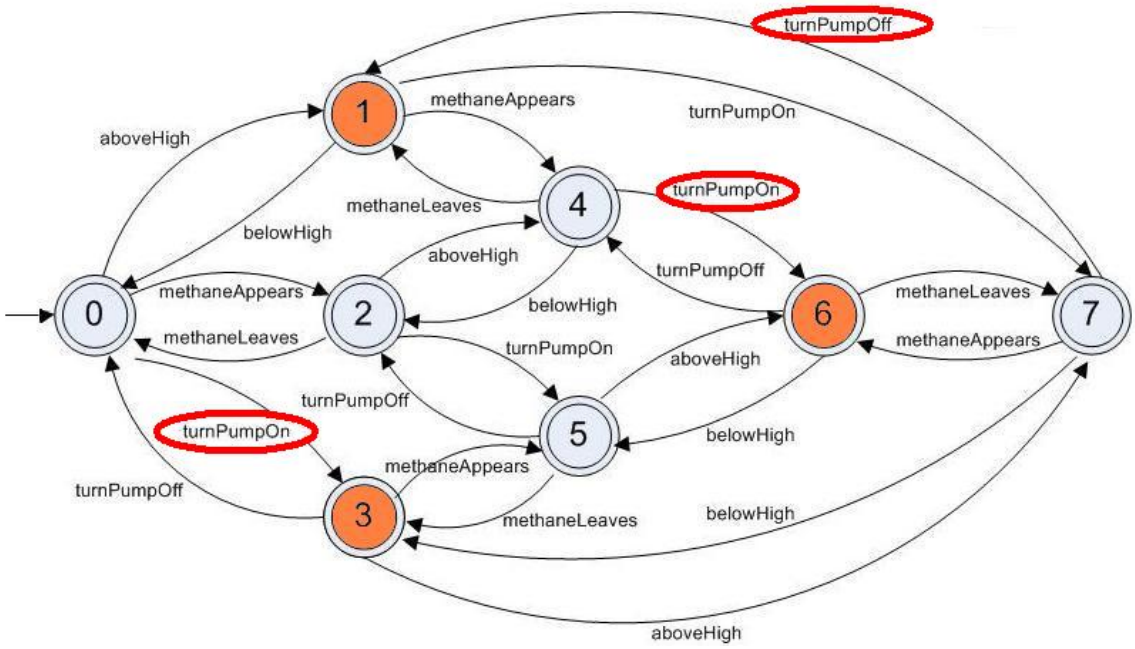

Figure 4.19: LTS for Simple Mine Pump (Initial stage)

the same fluent values) according to line 7 in the algorithm. Each of these goes through the *for*-loop on line 8 of Algorithm 4 to determine which paths in the general LTS we will eliminate according to their incoming transitions. States 1, 8, 9, and 10 in the negative PTA correspond to states 3, 6, 6, and 1 of the general LTS, respectively. According to the negative PTA in Figure 4.10, the incoming transitions into states 3, 6, and 1 labelled with actions *turnPumpOn*, *turnPumpOn*, and *turnPumpOff*, respectively, are unwanted, and should therefore be removed (line 11). This is illustrated in Figure 4.19.

However, doing this will not guarantee that *all* undesirable traces are removed from the general LTS, because there could be other unwanted traces present in the general LTS which have not been considered in the initial sample of negative scenarios, including those sharing the same prefix or suffix as one or more of these negative scenarios.

For example, the synthesised negative LTS shows that $\langle aboveHigh, methaneAppears, turnPumpOn\rangle$ is an undesirable trace, but from our knowledge of the system properties, we know that $\langle methaneAppears, turnPumpOn\rangle$ is also an example of undesirable behaviour. However, the negative PTA by itself does not consider this trace for pruning, and so it remains in the general

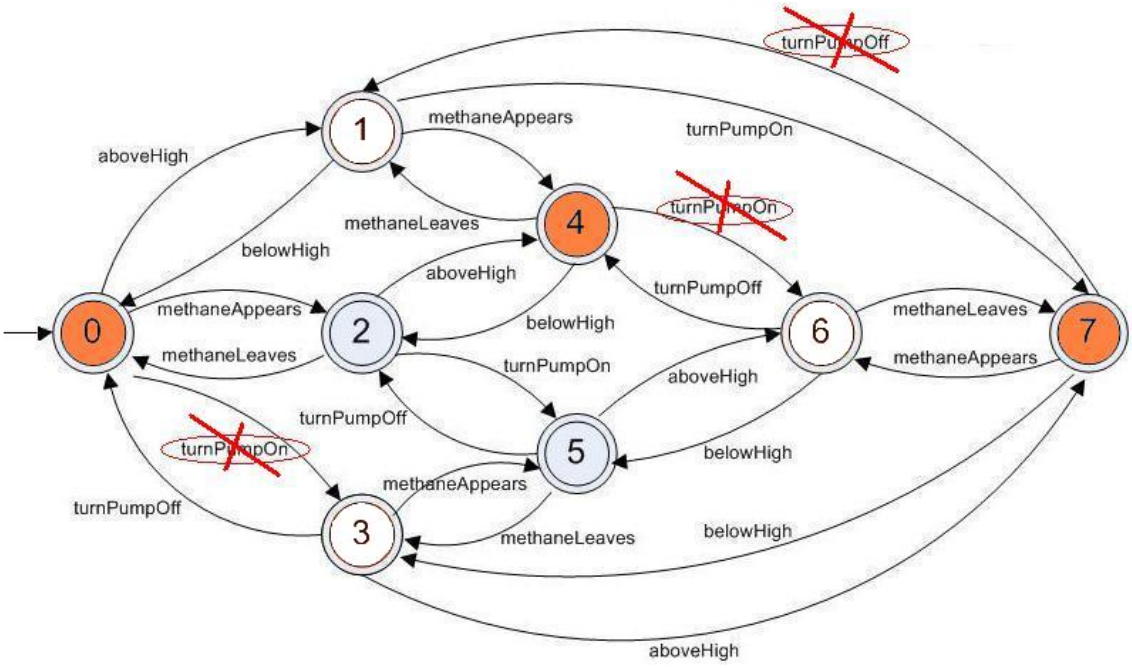

Figure 4.20: LTS for Simple Mine Pump - First-Level Pruning

LTS.

In order to eliminate such traces as well, we need to, in addition, analyse the states in the general LTS from which the unwanted transitions are outgoing (i.e. the previous states of the unwanted transitions), which explains line 9 of the algorithm. Using information regarding these states, we can then attempt to remove other transitions with the same action labels, by conducting further analysis of the states in the modified general LTS (line 13). This is explained in the next section.

4.4.2 Second-Level Pruning

Having eliminated three of the transitions in the general LTS, we now need to work with the states from which these transitions were outgoing, according to line 10 of Algorithm 4. In this case, we would need to look at states 0, 4, and 7 in the general LTS in Figure 4.17, as the unwanted transitions *turnPumpOn* and *turnPumpOff* (which have already been cut off) are outgoing from these states. Note that we no longer need to take into account states 1, 3 and 6 in the general LTS, as their unwanted incoming transitions

have already been dealt with, and consequently eliminated. The procedure is specified in Algorithm 5, which operates on the LTS shown in Figure 4.20.

Algorithm 5 Pruning: Phase II (a) - **FURTHER_PRUNE**

1: *INPUT: victims* = [(*ev*,*st*) | *ev* is the pruned transition event and *st* the source state of the pruned transition]
2: *OUTPUT:* refined system LTS *sys*

3: *sys* = system LTS
4: *pos* = generalised positive LTS
5: **for all** (transition t in *sys*) **do**
6: **if** (event e in t is such that (e,n) $\in$ *victims*) **then**
7: **if** (e matches a transition in *pos*) **then**
8: st = state in *pos* with outgoing transition e
9: s = source state in t
10: **if** (fluent value of st does NOT match that of s) **then**
11: **call** procedure **FLUENT_ANALYSIS**(s,*victims*)
12: **end if**
13: **else**
14: remove the outgoing transition e from s in *sys*
15: **end if**
16: **end if**
17: **end for**
18: **return** *sys*

Each transition in the given LTS is considered separately in order to decide whether it should remain in the system or should be eliminated, according to line 5 in Algorithm 5. However, to explain our approach in a structured way in this section, we will look at one event label at a time.

Let us first consider the event *turnPumpOn*. The transitions with this action label that have already been cut off are those outgoing from states 0 and 4 (this is taken into account at line 6 in Algorithm 5. As we can see in Figure 4.21, states 1 and 2 in the general LTS also have an outgoing transition labelled with the action *turnPumpOn*. In order to determine whether these can be considered for pruning or not, there are a number of different steps to be performed.

Firstly it is important to try and find a corresponding state with an outgoing transition labelled with *turnPumpOn* in the synthesised positive LTS in Figure 4.11, as we need to make sure that we do not unnecessarily remove a positive scenario by pruning the transition(s) in question. This

Algorithm 6 Pruning: Phase II (b) - **FLUENT_ANALYSIS**

1: *INPUT:* state s in sys whose outgoing transition e is being tested; $victims = [(e,st) \mid e$ is the pruned event from $st]$
2: *OUTPUT:* system LTS sys

3: $f =$ fluent values of s
4: $pos =$ generalised positive LTS
5: **for all** $((e,st) \in victims)$ **do**
6: $inter =$ intersection of fluent values in s and st
7: $power =$ power set of elements in $inter$, in ascending order of subset size excluding []
8: **for all** ($subset$ in $power$) **do**
9: **if** ($subset$ matches a state in pos with an outgoing transition e) **then**
10: **go to** line 8
11: **else**
12: remove transition e from sys
13: **break**
14: **end if**
15: **end for**
16: **if** (no state found in pos which contains fluent values in $subset$ **then**
17: **break**
18: **else**
19: **go to** line 5
20: **end if**
21: **end for**

22: **return** sys

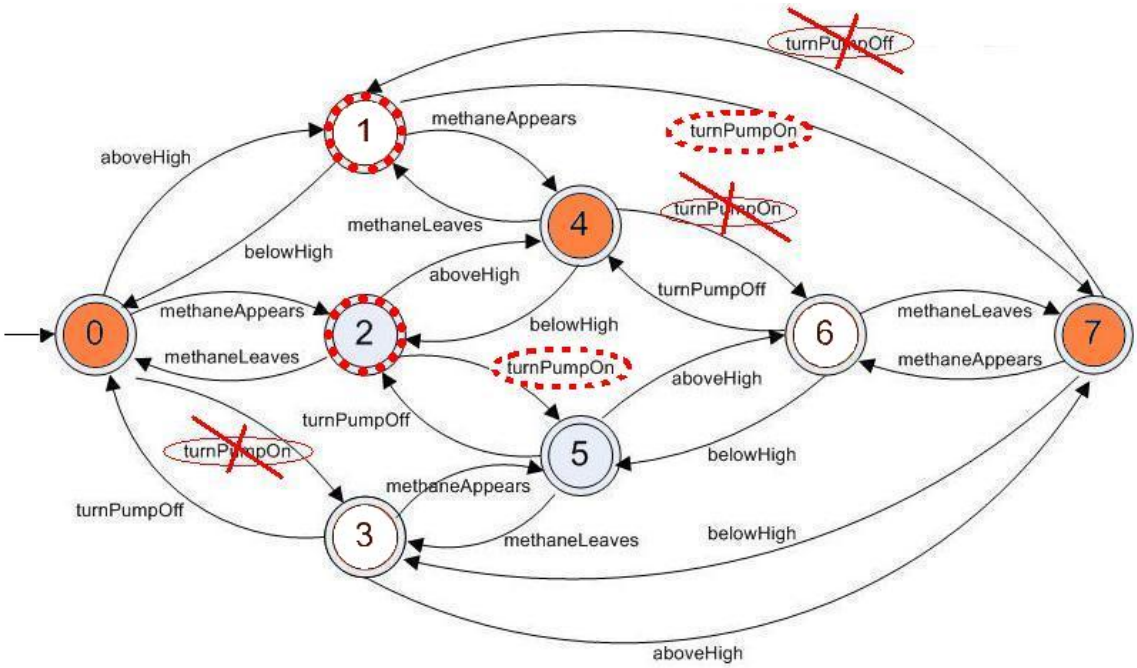

Figure 4.21: LTS for Simple Mine Pump - Second-Level Pruning (i)

takes place at line 8 in Algorithm 5. Already we can see that the transition outgoing from state 1 cannot be removed because Figure 4.11 shows that a state defined by the fluent *HighW* (such as state 1 in this case, or state {1, 2, 4, 9, 10} in Figure 4.11) can have an outgoing transition labelled *turnPumpOn*, and therefore does not satisfy the *if*-case on line 11.

This just leaves us with the outgoing transition from state 2 as a possible candidate for pruning, which we compare to other transitions in the initial LTS in order to decide whether its outgoing transition should remain or should be eliminated.

Fluent Conjunctions Method

This is the method which we have decided to use as part of the overall approach, because it performs an exhaustive analysis of fluents at equivalent states in the initial and generalised positive LTSs. We still consider fluent values within states, but this time we go further and devise sets of fluent conjunctions for each state, and then use intersections between these sets, together with the generalised positive LTS, in order to decide whether a particular transition should be pruned or not. This is closely linked to the Fluent-Set analysis that we investigated during earlier stages of the project, as explained in the Discussion subsection, and its pseudocode is available in Algorithm 6.

In the case of state 2, even though there is a state in the synthesised positive LTS in Figure 4.11 defined by the fluent values {*Methane,* ¬*HighW,* ¬*PumpOn*}, namely state 3, it only allows an outgoing transition with action label *methaneLeaves*, and none labelled *turnPumpOn*. The *if*-case at line 8 in Algorithm 5 is therefore not satisified, so according to line 15 the *turnPumpOn* transition outgoing from state 2 is pruned.

Now we can perform a similar analysis on the *turnPumpOff* transitions in the system LTS. As a result of the initial elimination of the outgoing *turnPumpOff* transition from state 7, which was eliminated because it covered one of the negative scenarios, we now need to consider other transitions with the same label, starting from states which have the same fluent values as state 7. These are states 3, 5, and 6 (Figure 4.22). According to their fluent values, state 3 corresponds to state {11,14} in the synthesised positive LTS in Figure 4.11, which shows an outgoing transition labelled *turnPumpOff* as an example of a desirable trace. Therefore, we can not consider the *turnPumpOff* transition outgoing from state 3 for elimination, as this would incorrectly get rid of desirable system behaviour (hence the *if-condition* at line 7 of Algorithm 5 is satisfied). States 5 and 6 do not have

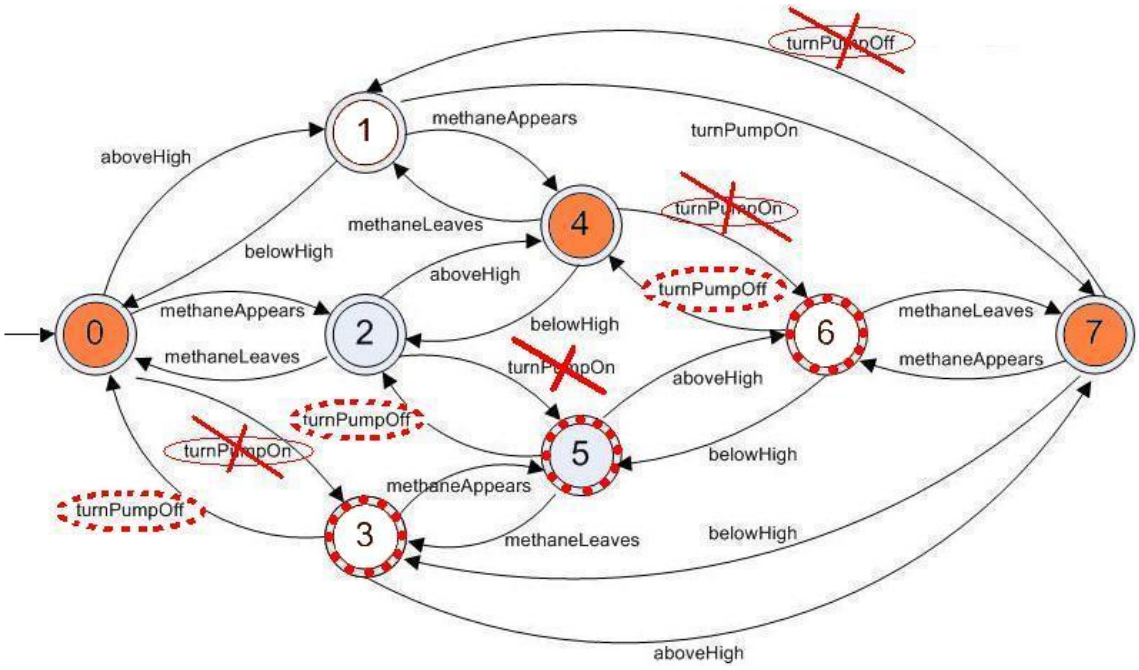

Figure 4.22: Refined LTS for the Simple Mine Pump - Second-Level Pruning (ii)

a corresponding state in the synthesised positive LTS, so we need to perform the fluent analysis specified in Algorithm 6.

We note the fluent values that state 7 has in common with states 5 and 6 (according to line 6) , and these are:

- $PumpOn$ (with state 5), and
- $\{HighW, PumpOn\}$ (with state 6)

The former case is straightforward, as we know after having dealt with state 3, which was defined by the fluent value $PumpOn$, that we cannot prune an outgoing transition labelled *turnPumpOff* from a state in which the fluent $PumpOn$ is true because it covers a positive scenario. The corresponding state in the positive LTS is state {11,14}. Hence, the outgoing transition from state 5 cannot be removed from the general LTS.

In the case of state 6, we would need to construct the power set of $\{HighW, PumpOn\}$, as specified in line 7, as these are the fluents that it has in common with state 7.

$$P(\{HighW, PumpOn\}) = \{\{HighW\}, \{PumpOn\}, \{HighW, PumpOn\}\}$$

We already know that the $\{PumpOn\}$ fluent by itself cannot be used to justify the pruning of the outgoing transition labelled with the action *turnPumpOff* from state 6, which would be consistent with the decision taken when dealing with states 3 and 5 (the pruning of their outgoing transitions with this action label was rejected because of an overlap with a positive trace). However, we can consider the fluent $HighW$. State {11,14} in the positive LTS is defined by the fluents {¬*HighW, PumpOn,* ¬*Methane*}. As we can see, the state does *not* include $HighW$ with a true value, and there is no other state in the model with an outgoing transition labelled with the same action (hence the *else*-case on line 11 is satisfied). It is therefore safe to justify the elimination of the outgoing transition from state 6, corresponding to line 12 of Algorithm 6, because of the presence of the $HighW$ fluent, which does not appear in the state in the positive LTS with the same outgoing transition.

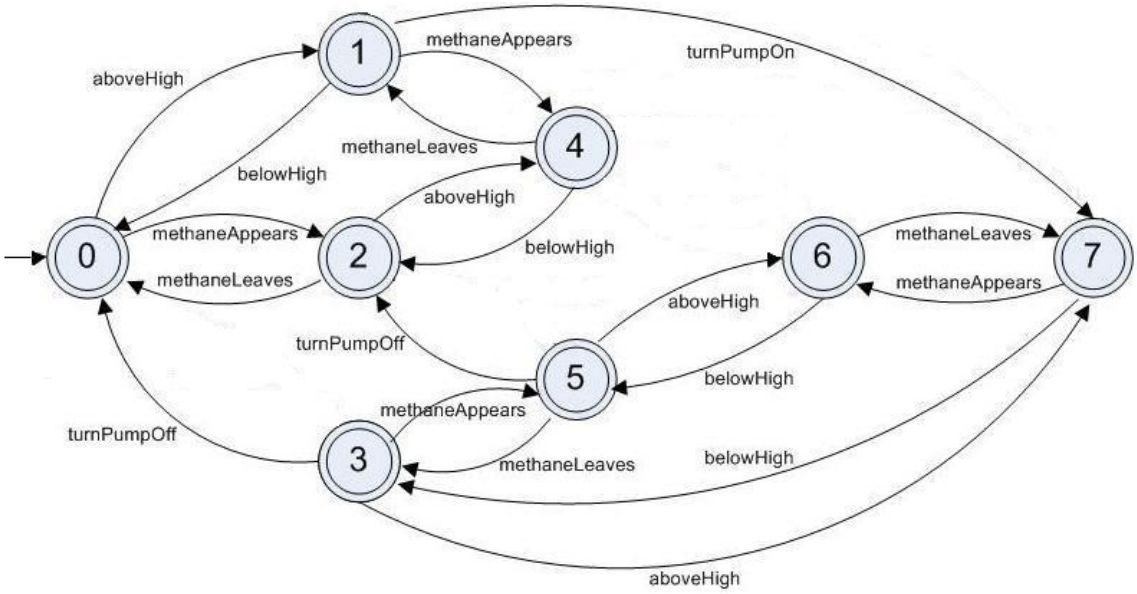

Figure 4.23: Most specific LTS for Simple Mine Pump - result using Fluent Conjunctions

The final outcome after all these steps is returned at line 22 of Algorithm 6, and is shown in Figure 4.23. We can see that all the undesirable behaviour conveyed through the initial sample of negative scenarios is no

longer present, whilst all the desirable traces denoted by the positive LTS are maintained in the new LTS.

Note that the results obtained need to be compared to the outcome of the ILP task, which is a more "greedy" algorithm that returns the *most* refined LTS possible for the given system. In this case, it produces the same result, as shown in Figure 4.24. The diagrams themselves may look different, but if we closely analyse the transitions in each of them, we see that they are equivalent.

4.4.3 Discussion

Maximum number of states and transitions in the initial LTS

The maximum number of transitions in the initial LTS model can be quantified. We assume that the same event cannot occur consecutively, hence we know that at any state, there will only be a finite number of outgoing transitions, one for each different event that can occur at the state in question. An informal proof follows.

The key factor that determines *which* events can take place at a particular state is the state's fluent values. As we have described earlier, fluents are defined by sets of initiating and terminating events which will cause the fluents to start or to stop holding, if and only if the fluent's value before the event occurrence was false or true, respectively. For example, the fluent definitions outlined previously for the system in Figure 4.17 show that the fluent *HighW* is initiated by the *aboveHigh* event and terminated by the *belowHigh* event. Thus, a state in which the fluent *HighW* is true can have an outgoing transition with the label *belowHigh* - which will stop the fluent from holding -, but not an *aboveHigh* event because this would mean initiating a fluent that has *already* been initiated previously, and therefore currently holds! It would therefore be a pointless action.

For the LTS in Figure 4.17, which has 6 different events and 3 fluents, this means that every state can have 3 outgoing transitions, and 3 incoming ones. Notice that this does add up to the total number of events, and hence shows that in the most general LTS, *all* states undergo the effects of each and every system event, some in the form of incoming transitions and others in the form of outgoing ones.

We have shown that there are a finite number of transitions at every state, and according to our initial assumptions, each state is unique in terms of its fluent values. Thus the maximum number of states in the most general LTS is purely determined by the number of fluents in the system. Because

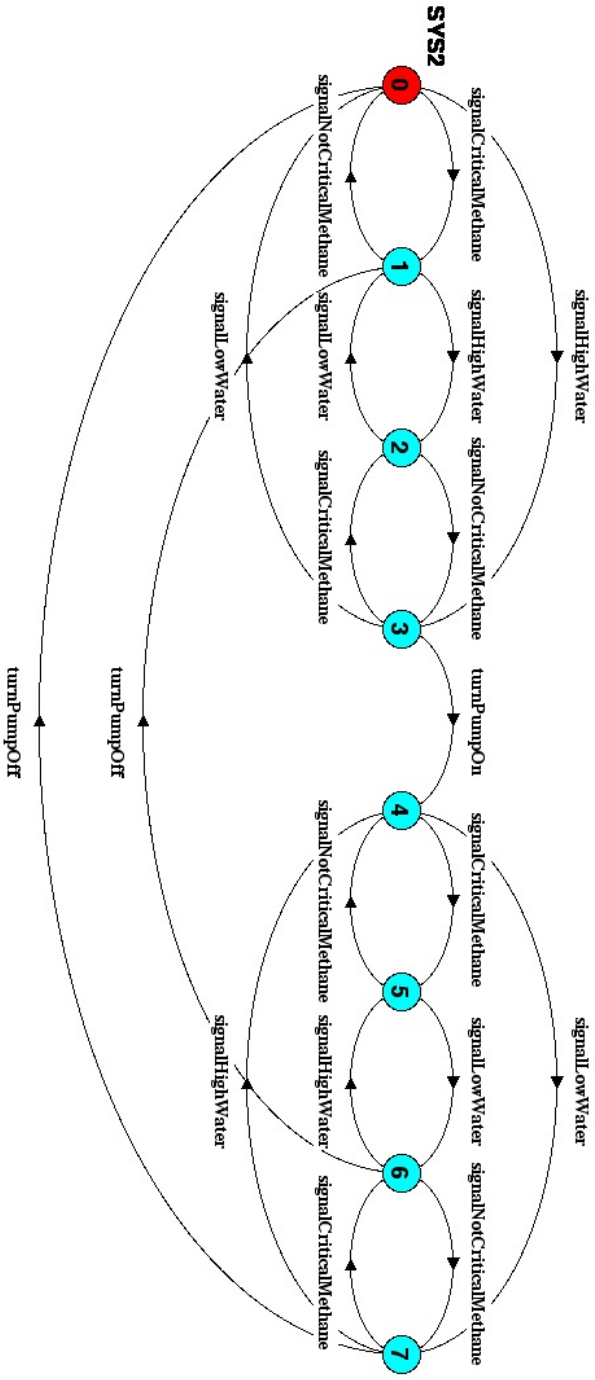

Figure 4.24: Results of ILP approach

every fluent can take one of two values, namely true or false, this means that for a system with n fluents, the total number of states in the most general LTS is 2^n. Checking this with our simple Mine Pump Control system which has 3 fluents, we can see in Figure 4.17 that there are a maximum of 8 states, which correctly corresponds to 2^3.

At the initial stage, only the domain conditions have been taken into account, in order to avoid the same transitions from occurring consecutively. This also avoids outgoing transitions that are labelled with an action that terminates (resp. initiates) a certain fluent, from a state where that fluent does not already hold (resp. already holds). For instance, for the LTS in Figure 4.17, there cannot be a transition labelled *turnPumpOff* from state 0 because at this state the *PumpOn* fluent does *not* hold (i.e. the pump is already off).

Fluent-Set Analysis using Positive and Negative Scenarios

This section briefly describes our very first ideas related to the analysis of fluent values in particular states. In this case we consider the fluents holding at the state immediately before the transition that is under examination. Despite having a certain level of abstraction, this forms the basis of our intuition, which then led us to take the idea further and to look at how this could be utilised in the final refinement on the general system LTS.

For each unwanted transition in the negative PTA entailed by a negative scenario, we thought it is sensible to study the fluents holding at states which have these unwanted outgoing transitions. Note that working with the states in this way rather than with specific traces introduces certain flexibility regarding the order in which events may occur. Hence, this way we consider *all possible permutations* of a particular set of events, so that in most cases the exact order in which these occur does not matter in order to identify paths within the initial LTS that are desirable or not.

For instance, for the Simple Mine Pump system, let us consider the *turnPumpOff* event, which refers to the action of turning the pump off. According to a given set of positive and negative scenarios, we can classify states with an outgoing *turnPumpOff* event according to their fluent values, as below. Note that *positive* states can be any of the states in the positive PTA, or any non-final states of the negative PTA. The latter is true according to our definition of *accepting states* provided in Section 4.2. On the other hand, negative states are final states in the negative PTA.

1. Examples of states *accepting* a *turnPumpOff* event, referred to as *pos-*

itive states are those where the following fluents hold:

- $\langle PumpOn,\ \neg HighW,\ \neg Methane \rangle$
- $\langle PumpOn,\ HighW,\ Methane \rangle$

2. Examples of states *rejecting* a *turnPumpOff* event, referred to as *negative states* are those where the following fluents hold:

   - $\langle \neg PumpOn \rangle$
   - $\langle PumpOn,\ HighW,\ \neg Methane \rangle$

By considering each set of fluents as a conjunction of those fluents, we can say the following: a *turnPumpOff* event is an example of *desirable* system behaviour iff the state with the outgoing *turnPumpOff* event is defined by the fluent values

- $\langle PumpOn \wedge \neg HighW \wedge \neg Methane \rangle$ or
- $\langle PumpOn \wedge HighW \wedge Methane \rangle$

Similarly, a *turnPumpOff* event is an example of *undesirable* system behaviour iff the state with the outgoing *turnPumpOff* event is defined by the fluent values

- $\langle \neg PumpOn \rangle$ or
- $\langle HighW \wedge \neg Methane \wedge PumpOn \rangle$

The overall objective is to use these results when analysing states in the initial LTS which have an outgoing *turnPumpOff* transition, by identifying (conjunctions of) fluents that hold at these states. Consequently, we can decide whether to prune these transitions or not.

One way would be by checking if there are any fluent values that are only found in the negative states. If there exists a set F of such fluents, then *all* the *turnPumpOff* transitions outgoing from states in which one or more fluents in F hold, could be pruned from the initial LTS. For example, with the example states provided above, we can see that the fluent value $\neg PumpOn$ *only* occurs in the negative states; both the positive states have the value *PumpOn*. Therefore any *turnPumpOff* transition outgoing from a state whose fluent values include $\neg PumpOn$, would be pruned from the initial LTS.

However, it may not always be the case that there are fluents that appear only in the negative states, so to ensure we do not perform "greedy" learning which could over-constrain the system by eliminating some of the positive behaviour as well, it is necessary to construct the set of *all* fluents that are encountered over the entire set of positive states, namely the union

$$\bigcup_{E^+} = \{PumpOn,\ \neg HighW,\ HighW,\ \neg Methane,\ Methane\}$$

These can be compared to the set of fluents encountered with the negative scenarios, namely the union

$$\bigcup_{E^-} = \{PumpOn,\ \neg PumpOn,\ HighW,\ \neg Methane\}$$

However, since we have already considered $\neg PumpOn$ as a fluent value that only holds amongst the negative states, we can eliminate it from the set to get:

$$\bigcup_{E^-} = \{PumpOn,\ HighW,\ \neg Methane\}$$

The modified $\bigcup_{E^-}$ set now shows the fluents which occur both in the negative scenarios and the positive ones. It would therefore be of no use to look at these individually and compare them to $\bigcup_{E^+}$.

However, we can construct *conjunctions* of the fluents in $\bigcup_{E^-}$, and check their occurrence in any of the positive states. If a conjunction does *not* appear in any of the positive states, then it would constitute another way of rejecting unwanted *turnPumpOff* transitions from the initial LTS. The conjunctions are:

- $\langle HighW \wedge \neg Methane \rangle$
- $\langle HighW \wedge PumpOn \rangle$
- $\langle \neg Methane \wedge PumpOn \rangle$
- $\langle HighW \wedge PumpOn \wedge \neg Methane \rangle$

The second and third items in the list above fail the necessary criteria, as these conjunctions *do* occur amongst the positive states. The first element of the list, however, does not occur in any of the positive states, which means that we can eliminate the *turnPumpOff* event outgoing from any state in the initial LTS where the fluents *HighW* and $\neg Methane$ hold. Similarly, the last conjunction in the list above does not violate any of the desirable

behaviour expressed through positive states. In fact, it coincides with one of the negative states, and can therefore be used to eliminate further traces.

As we can see, these (conjunctions of) fluents help identify states in the initial LTS which have undesirable outgoing *turnPumpOff* transitions, and should therefore be eliminated. Using this sort of fluent-analysis together with the system's domain pre-conditions, the resulting LTS can reject the negative behaviour of the system in question whilst accepting all the positive behaviour. Hence, a similar approach has been taken in our *Fluent Conjunctions* method that performs the second-level pruning in our refinement algorithm.

Equivalent States Method

After arriving at the system shown in 4.26, this is the alternative method that we considered to further analyse the transitions that need to be eliminated from the system LTS, using a comparison between states and transitions. For a given state in the system LTS, the pruning criteria used to remove its outgoing transition is illustrated in Figure 4.25.

Following completion of the *first-level pruning* phase of our refinement algorithm, this method would look for a state corresponding to state 2 from the general LTS, in the synthesised positive LTS in Figures 4.27. We can see that state 3 in the positive PTA is defined by the same fluent, however, it only allows an outgoing transition with action label *methaneLeaves*, and no transition with label *turnPumpOn*. According to the flowchart in Figure 4.25, we would therefore need to find an equivalent state in the negative PTA. Once again, state 3 in the negative PTA is a corresponding state, which only allows an outgoing transition labelled *aboveHigh*. This does not provide us with any further information to aid the synthesis process, but at the same time does not suggest that if we removed the outgoing transition with this label from state 2 in the general LTS, then we would be removing desirable behaviour. Hence, the trace $2 \stackrel{turnPumpOn}{\longrightarrow} 5$ is eliminated since this method of refinement is a more "greedy" approach.

Now let us consider the action *turnPumpOff*. The outgoing transition from state 7 in the general LTS labelled with this action has already been eliminated, but we now need to consider other transitions with the same label, starting from states which have the same fluent values as state 7. These are states 3, 5, and 6 (see Figure 4.28). According to their fluent values, state 3 corresponds to state {11,14} in the synthesised positive LTS in Figure 4.11, which shows an outgoing transition labelled *turnPumpOff* as an example of a desirable trace. We can therefore not consider the *turnPumpOff*

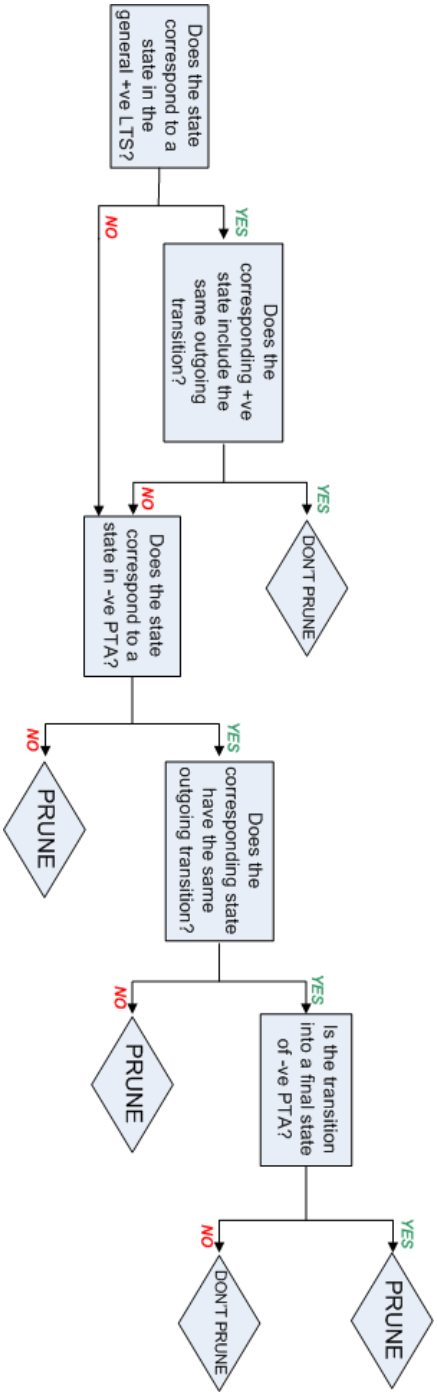

Figure 4.25: Equivalent States Method

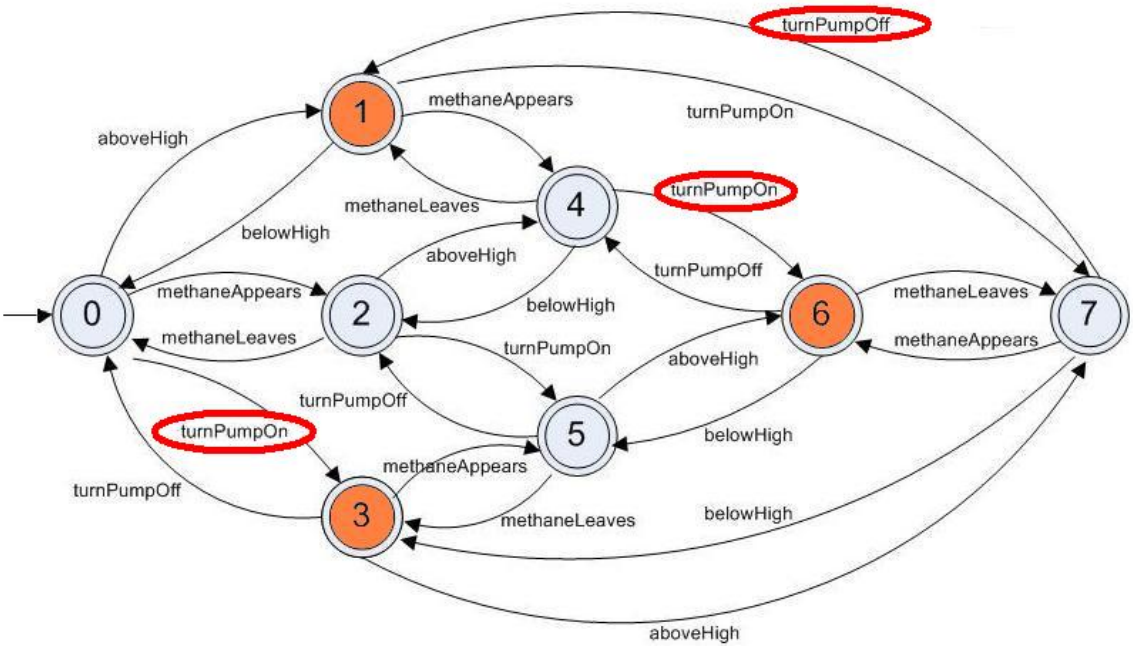

Figure 4.26: LTS for Simple Mine Pump (Initial stage)

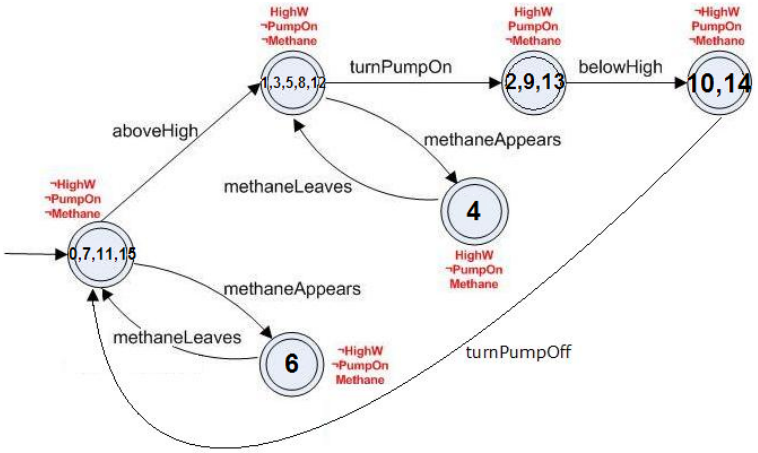

Figure 4.27: Synthesised LTS for positive scenarios

transition outgoing from state 3 for elimination, as this would incorrectly get rid of desirable system behaviour.

States 5 and 6 do not have a corresponding state in the synthesised pos-

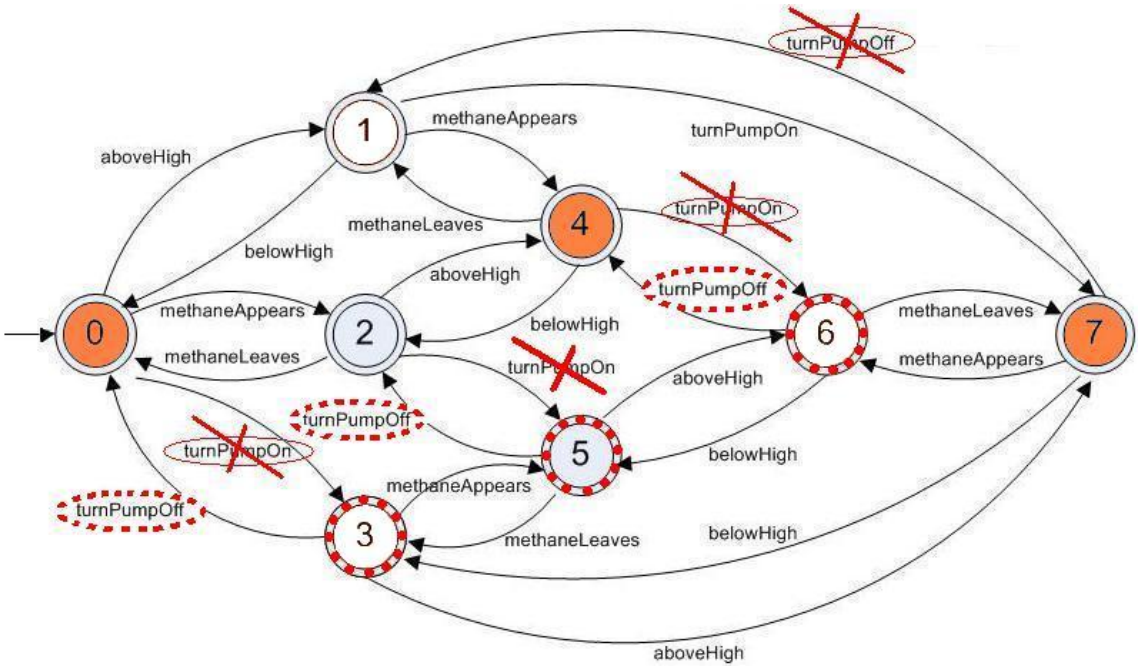

Figure 4.28: Refined LTS for the Simple Mine Pump - Equivalent States Method (i)

itive LTS, but they do have equivalent states in the negative PTA. In the case of state 5, the equivalent state is state 7, which enables a *turnPumpOff* transition. However, the transition goes into a final state, and hence represents undesirable behaviour. Hence it suggests that the transition outgoing from state 5 in the general LTS should be pruned.

As for state 6, the equivalent states in the negative PTA in Figure 4.10 are states 8 and 9. However, these are final states and therefore do not have any outgoing transitions. The lack of information therefore does not suggest that by removing the transition from state 6 in the general LTS, we would be eliminating any desirable behaviour. Hence, the greedy approach in this case decides to eliminate the transition.

The final outcome after all these steps is shown in Figure 4.29, where all the undesirable behaviour conveyed through the initial sample of negative scenarios is no longer present, whilst maintaining all the desirable traces denoted by the positive LTS.

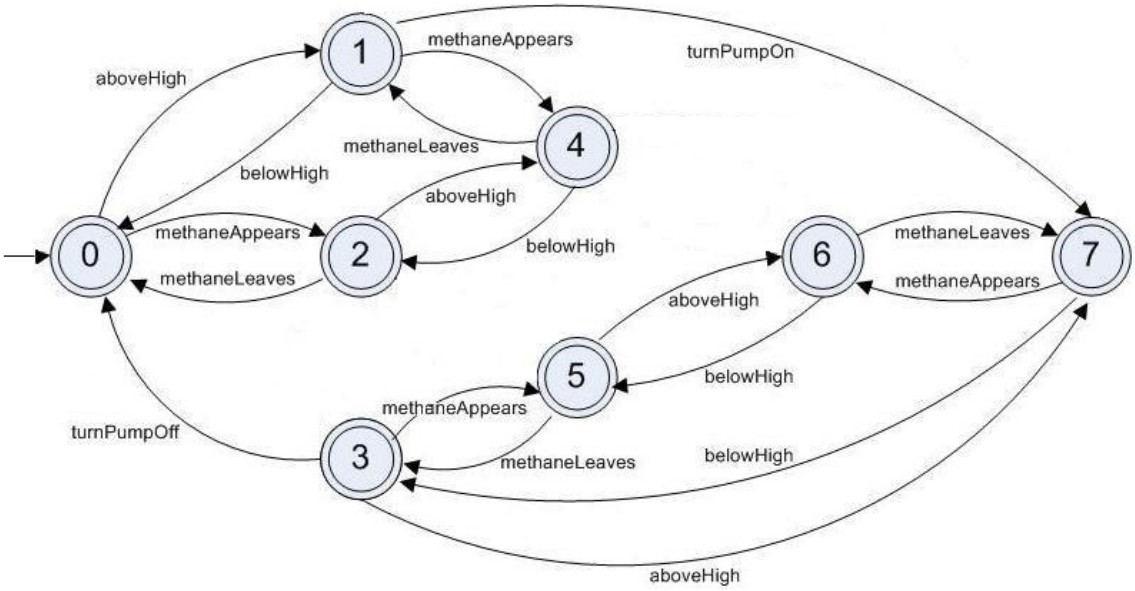

Figure 4.29: Most specific LTS for the Simple Mine Pump - Equivalent States Method (ii)

Chapter 5

Comparison to Related Work

Many of the approaches presented in Chapter 2 share some common disadvantages, that our approach tries to avoid. Amongst other requirements, the various efforts do *not* always consider both positive *and* negative examples, and they may require additional input from the users. Many of the results they produce are not easily understandable.

In some cases, end-users have to provide pre- and post-conditions of scenario interactions, which is not always possible. In addition, this information may need to be refactored during further stages when more positive or negative scenario examples are provided during synthesis. The more interactive techniques can solve this problem as they do not need any additional input besides the specification and some scenarios, but this also implies that a large number of questions may need to be posed to end-users in order to obtain the necessary information. This would require a lot of effort from the user, and would increase the burden on them as well as making us more dependent on their input. A common problem that we often experience while synthesising behavioural models is that of *overgeneralisation* (resp. *undergeneralisation*). The former occurs when the resulting model incorrectly covers some of the negative scenarios, and the latter implies that some of the positive scenarios are incorrectly rejected.

The principal factor differentiating the previous methodologies from our task is that they work towards the *generation* of a model starting from a certain input, whereas our objective is to work on an *existing model* (in this case an LTS), and to define an algorithm for refining the given model using scenario-based specifications.

5.1 Generalisation of LTSs from Scenarios

The technique described in [8], for instance, does *not* take into account negative scenarios, unlike its automated counterpart proposed in [9]. Note that in this case the target is to ensure that the new examples (paths in the automaton) added by users do not cover any undesirable behaviour of the system, which is slightly different to our aim of eliminating paths whilst making sure that all the previously accepted desirable behaviour is still being covered.

As mentioned previously, the state-merging algorithm that forms part of our refinement approach is based on [8], which uses this technique to generalise system behaviour. Thus, it attempts to overcome problems such as the need for users to provided additional input, and the difficulty in understanding the generated LTSs.

However, the synthesised model in [8] can be prone to inconsistencies if different users submit scenarios that are not consistent with each other, and this is a problem that our approach could face as well if the users providing the negative scenarios are different to those providing the positive ones.

[8] only deals with existential scenarios that state what *may* occur, instead of covering universal scenarios as well, which state what *must* occur. Similarly, our approach works on an LTS defined by domain knowledge purely, and thus does *not* consider system goals and *trigger* conditions, which would have helped state the system's compulsory behaviour.

However, both the approaches mentioned in [8, 9] are sensitive to classification errors, as the implications of accepting a scenario rather than rejecting it can be very costly at the synthesis phase - a problem that is common to most learning-by-examples techniques. Our approach tackles the problem in a slightly different manner, since we assume that the positive and negative scenarios provided at any particular refinement stage are already covered by the given LTS. Hence it is *not* possible that a scenario that was rejected at a previous refinement stage (and hence eliminated from the given LTS), is classified as an example of desirable behaviour at a later stage.

In contrast to the bottom-up approach presented in [8, 9], we intend to use our approach to learn a language L whilst maintaining the alphabet A, hence solving the problem in a top-down fashion. This would imply the specialisation of the initial automaton (rather than generalisation) into a different automaton that would incrementally recognise a subset L^* of the language, i.e. $L^* \subset L$.

5.2 Using ILP to Extract Operational Requirements from Scenarios

The inductive learning approach presented in [2, 3] also tries to learn a system specification from given sets of scenarios. Starting with the most general LTS as input, the learning task delivers the most specialised model which rejects *all* the given negative scenarios whilst still covering all the positive scenarios. As we can see, this approach and our refinement algorithm aim to achieve the same objective, and hence share a number of similarities.

The ILP approach uses the notions of abduction, deduction, and induction in order to learn Horn clauses from the given sets of scenarios, which are then used to extend the given system specification. Our implementation encompasses the latter two, namely the notions of deduction and induction. In our methodology, the deductive phase can be referred to as the one which determines the boolean value of a fluent at any state. As described previously, fluents can start to hold or stop holding depending on the occurrence of initiating and terminating events with respect to those fluents. Therefore, given an event, we can deduce whether the value of a particular fluent is true or not following that event. We are also borrowing the idea of checking only those properties that are *true* at any state, as these represent pre-conditions to the execution of certain events. This constitutes our inductive phase.

In addition, note that our approach can be related to some of the axioms that comprise the requirements specification (Chapter 2) used by the ILP approach. The initial-state, persistence and change axioms help define what would be the most general LTS in our approach. On the other hand, event precondition axioms form the principal component of the learning process, as they determine how the initial LTS is modified to return a more refined model.

However, the underlying methodologies used by the ILP task and our approach differ slightly in a number of ways. In contrast to our approach, the learning task presented in [2, 3] extracts operational requirements from partial system specifications through the use of event pre-conditions and trigger-conditions. Therefore, this method represents behavioural models in a declarative manner through the use of logic programs, which our approach bypasses.

Unlike the ILP approach, ours does not need to go through a translation process from LTL to EC before the learning task can occur. Hence we may experience an improvement in computation time when using our approach instead of the ILP task, but in addition our approach is able to work directly

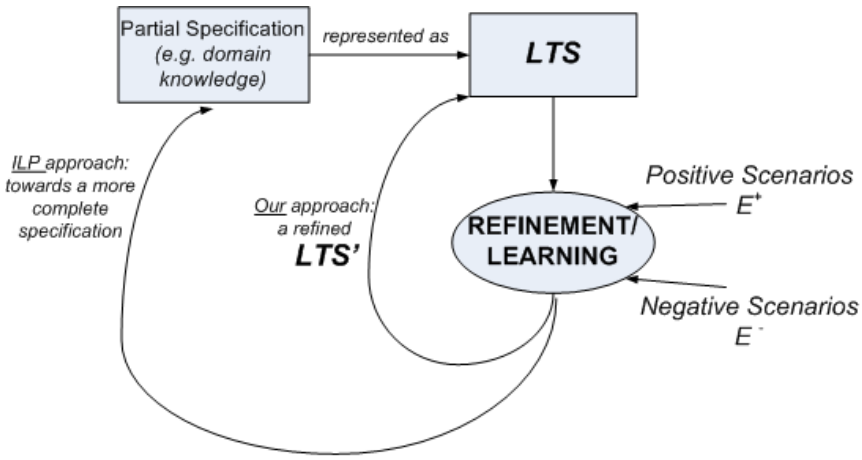

Figure 5.1: Comparing our approach to ILP

on the given LTS model.

The comparison between both approaches can be illustrated as in Figure 5.1. Both methods undergo a process of learning system properties implicitly expressed by the given sets of positive and negative scenarios. The difference lies in that the ILP uses the results from the learning task in order to extend the initial specification with the learnt *constraints*. The updated specification can then be used to generate a new LTS model corresponding to the system, and the learning task can take place once again. On the other hand, our refinement approach modifies the given LTS, and outputs a more refined model that can undergo further stages of refinement as long as there are examples of negative behaviour being covered by the system.

5.3 LTS Refinement based on Partition-Refining

[10] proposes a partition-refining algorithm that helps obtain the minimal canonical automaton corresponding to a given initial PTA, which is what our approach intends to achieve during the generalisation of given positive behaviour. It may not be the most efficient way of going about the specialisation of given models, but it does avoid the execution of separate minimisation steps after the induction phase, hence making it more efficient in practice for many cases.

Its application scope ranges to various different regular languages, such as those comprised of sets of strings, and pure automata in general. Hence

the method does not need to take into consideration additional knowledge such as system pre-conditions and domain conditions. In contrast, our algorithm is to be applied more generally to include such cases, taking into account fluents and events. The difference lies in the nature of both the problems, as we are not looking at sets of strings, but instead, we are working with events. In the latter case, we cannot usually have consecutive instances of the same event, whereas with characters and/or strings, this is acceptable, thus the number and type of possible paths in both systems will tend to vary.

In addition, the methods in [10] are of limited use to us because they considers positive examples *only*, whereas our aim is to take into account both positive and negative scenarios for a given behavioural model in order to correctly synthesise it.

5.4 Generalising Scenarios into State Charts

A similar merging algorithm to the one we use is described in [36], which represents scenarios using sequence diagrams instead. Similarly to the way in which we represent each scenario as a separate branch of a PTA, the approach in [36] translates each scenario into a separate FSP using semantic information. This results in the generalisation of behaviour such that user-defined requirements are satisfied, which is equivalent to our approach since we use scenarios provided by the users.

In addition, the approach in [36] analyses entire traces during the merging process, which is similar to our algorithm. Even though we merge states, our state-merging algorithm only allows a merge in the positive PTA to occur if the resulting *traces* do not cover any undesirable traces.

However, nodes in the different FSPs are merged according to system domain knowledge, due to which the final model is just an approximation of the system and therefore needs to be manually reviewed and modified. Our state-merging algorithm, on the other hand, uses given sets of positive and negative scenarios, so no further modifications to the generalised result need to be made by the user.

As with some of the other related approaches explained in Chapter 2, the algorithm in [36] focuses on the *generation* of system specifications using scenarios, whereas we can bypass such a stage and can directly *modify* the given LTS using our approach.

Nevertheless, [36] addresses some of the assumptions that we make in our approach, namely that the given sets of positive and negative scenarios

are already covered by the given LTS. In reality this may not be the case, so the approach in [36] detects conflicts in the domain knowledge to refine the given scenarios.

Chapter 6

Tool for the Refinement of LTSs

This chapter describes the implementation of the LTS refinement algorithm developed in this project. The implementation is a plug-in of the LTSA tool, and as such it has been implemented in Java.

We have used the LTSA because it is a tool used to analyse labelled transition systems and could therefore benefit from the addition of such a functionality, as discussed in Chapter 2.

6.1 Design & Implementation

As mentioned earlier, the algorithm we propose in this report performs the following 3 stages, given an initial LTS with examples of positive and negative scenarios covered by this LTS.

1. Representation of the input scenarios as PTAs
2. Generalisation of the positive PTA
3. Refinement of the given LTS

The overall structure of the added code can be observed in Figure 6.1, and it has been added to the existing source code for the LTSA tool as an additional package called *refinement*.

In this section we aim to describe the design choices made during the implementation of each stage, together with the reasoning behind these decisions. We will use our running example of the Simple Mine Pump to explain various features.

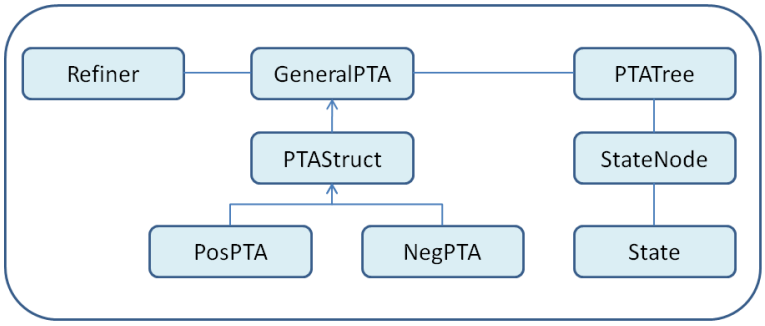

Figure 6.1: UML diagram for the *refinement* package

Collection of system data

For a given LTS, The LTSA tool performs parsing on the corresponding FSP. We use these results to populate a number of data structures which are used throughout our algorithm, some of which are also present in the original LTSA source code.

The sets of events and fluents for the system in question are stored in arrays, as shown in Figures 6.2 and 6.3.

Figure 6.2: Array of System Events

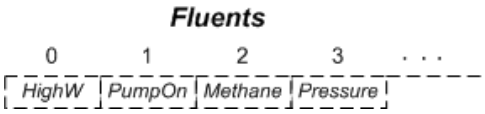

Figure 6.3: Array of System Fluents

The fluent definitions (in terms of sets of initiating and terminating event as defined in Chapter 2) are used to populate a matrix of events by fluents, showing how events influence the value of certain fluents (integer values are used, so 1 is used where an event initiates a fluent, -1 is used when an event terminates a fluent, and 0 is used where an event makes no difference to the value of a fluent). An example is shown in Figure 6.4. Note that the indices for events and fluents coincide with those in Figures 6.2 and 6.3. This makes it easy to use the indices throughout the code, as this us allows us to use the indices instead of the actual objects, and we can easily extract the value at an index whenever necessary.

We also store the fluent values of the initial state in the given LTS, as these will be used when we represent the given scenarios using defined structures. More details follow.

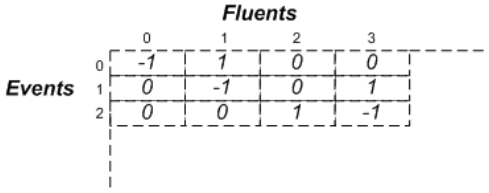

Figure 6.4: *Events* × *Fluents* Matrix

Representation of scenarios as PTAs

Let us consider the same sets of scenarios as those used previously. These are the following positive scenarios,

- $\langle signalHighWater,\ turnPumpOn \rangle$
- $\langle signalHighWater,\ signalCriticalMethane,\ signalNotCriticalMethane \rangle$
- $\langle signalCriticalMethane, signalNotCriticalMethane,$ $signalHighWater, turnPumpOn, signalLowWater, turnPumpOff \rangle$
- $\langle signalHighWater,\ turnPumpOn,\ signalLowWater,\ turnPumpOff \rangle$

and negative scenarios,

- $\langle turnPumpOn \rangle$
- $\langle signalHighWater,\ signalCriticalMethane,\ turnPumpOn \rangle$
- $\langle signalHighWater,\ turnPumpOn,\ turnPumpOff \rangle$
- $\langle signalCriticalMethane,\ signalHighWater,\ turnPumpOn \rangle$

We assume that users provide scenarios as text files. Two separate files are needed to represent the sets of positive and negative scenarios, respectively, and in both cases, scenarios are written as sequences of events in the following format, each scenario on a separate line:

$$e_0 > e_1 > ... > e_n$$

where $\forall i$: $0..n$, each e_i is an element of the global set of system events.

Note that the representation of both positive and negative scenarios is identical, which means we could have chosen to accept a single file that includes *all* the given scenarios. Separate delimiters would then be needed

to differentiate between positive and negative scenarios, hence requiring the need for additional parsing. In order to avoid this, we have decided to keep the different types of scenarios separate from each other, hence making it easier to access each set as and when required. In addition, we thought this is more user-friendly, and it may even avoid incorrect classification of scenarios by the users.

Analogous to the LTSA representation of the given LTS, we thought it is appropriate to represent the information contained within the scenario text files using a similar data structure. Thus, we can use them simultaneously during various stages of our refinement algorithm, without the need to perform extensive changes between the underlying structures.

Each given set of scenarios is contained within an ArrayList object of String arrays, where each String array corresponds to a separate scenario. Scenarios are represented as tree objects called *PTATree*, where each tree is represented by a set of *StateNode* objects denoted by the index of the state that they represent, as well as information about the state itself, such as its fluent values. Each node stores information regarding its parent and children nodes. The number of nodes for the PTA is calculated using the length of the arrays representing the scenarios, plus 1 for the root node, since PTAs have a common root node out of which different branches emanate to portray each scenario. For example, let us consider the first positive scenario:

$$\langle \textit{signalHighWater, turnPumpOn} \rangle$$

Since we assume that all scenarios start at the same initial state, this means that the node corresponding to the initial state has a number of outgoing transitions that is the same as the number of scenarios. The event labels for each of these transitions is the *first* event from each scenario. For the example scenario provided above, there is an outgoing transition from the initial state labelled with *signalHighWater* to another state. The latter state in turn has an outgoing transition labelled *turnPumpOn* to another state. Therefore, 2 different nodes are created for this scenario, and if this is done for each scenario, then the total number of state nodes in the final structure will be 1 plus the the length of each scenario.

For instance, the set of positive scenarios listed at the beginning of this section would be represented as the following:

- $0 \overset{signalHighWater}{\rightarrow} 1 \overset{turnPumpOn}{\rightarrow} 2$
- $0 \overset{signalHighWater}{\rightarrow} 3 \overset{signalCriticalMethane}{\rightarrow} 4 \overset{signalNotCriticalMethane}{\rightarrow} 5$

- $0 \xrightarrow{signalCriticalMethane} 6 \xrightarrow{signalNotCriticalMethane} 7 \xrightarrow{signalHighWater} 8 \xrightarrow{turnPumpOn} 9 \xrightarrow{signalLowWater} 10 \xrightarrow{turnPumpOff} 11$
- $0 \xrightarrow{signalHighWater} 12 \xrightarrow{turnPumpOn} 13 \xrightarrow{signalLowWater} 14 \xrightarrow{turnPumpOff} 15$

For each transition between a pair of states, a *Transition* object is created of the form:

(*source, event, target*)

where *source* is the state from which the transition is *outgoing*, *event* is the event label for this transition, and *target* is the state which has this transition as an incoming one. The representation coincides with the one used in the original source code. It is then added to an ArrayList of transitions corresponding to the PTA structure.

There is a slight difference between the structures that we use in the *refinement* package to represent states, compared to those used by the existing LTSA source code. Firstly, our *State* objects contain information that is relevant to the state itself, such as its accepting/non-accepting status, and corresponding fluent values.

The former is a private field that we have added in order to differentiate states in the negative PTA from those in the positive PTA. We remind ourselves that *all* states in the positive PTA are accepting, since a trace ending at any of the states is considered as a system execution. However only the *final* states of a negative PTA are accepting because prefixes of the negative traces contained within the negative scenarios can be acceptable traces, but it's only the last transition of every branch which determines the undesirable behaviour. This information is necessary during later stages of the refinement process, as explained later in this chapter.

The *StateNode* corresponding to a system state stores information regarding a node's predecessors and successors. These are the nodes that are connected to the node in question by a labelled transition, i.e. one-step-reachable states. A node's parent or child node is represented as a (*node*,*event*) pair, where *node* is the parent (resp. child) node, and *event* is the label of the transition between the two nodes. Such a tuple is present for each parent (resp. child) node, and is stored in the corresponding map. For example, for state 0 corresponding to the scenarios listed above, the map of parent nodes would be empty, since the state has no parents, but the map of children nodes would include:

{(1, *signalHighWater*), (3, *signalHighWater*), (6, *signalCriticalMethane*), (12, *signalHighWater*)}

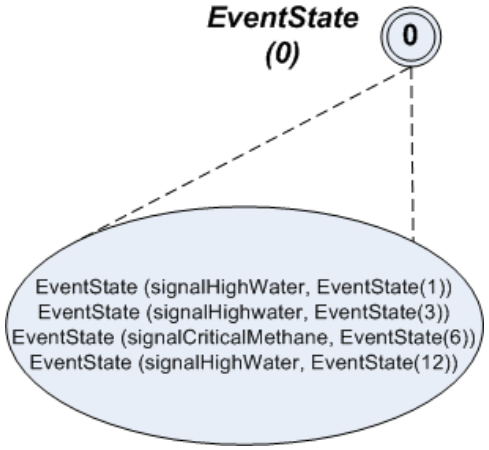

Figure 6.5: LTSA representation of a state

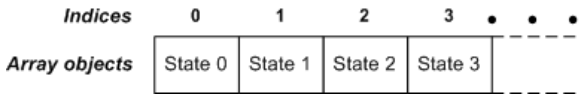

Figure 6.6: Our states array

On the other hand, the existing LTSA representation of a state is an *EventState* object. There is a separate *EventState* for each transition in the given system. Hence, for a state with multiple outgoing transitions such as state 0 in the example provided, there would be multiple *EventState* objects as shown in Figure 6.6.

Once both the positive and negative scenario trees have been populated, the states are added to an array which stores *State* objects at indices that coincide with the state's index, as shown in Figure 5.

Using information regarding the fluent values at the initial state of the given LTS, together with the ArrayList of transitions and the matrix which shows the different fluent definitions such as the example in Figure 6.4, we can determine the fluents that hold at each state.

For example, using knowledge of the fluents that hold at the initial state, we can determine the fluents that hold at each of its children states. This is determined by using the fluent definitions array such as those in Figure 6.4, which conveys information regarding the fluents that hold (resp. do not hold) as a result of certain events. Using this we can determine which fluents will hold at the children states of state 0 depending on its outgoing

transitions. This is repeated recursively for all other states. As a result, we can populate the two other private fields corresponding to State objects, namely sets which contain their true fluents and false fluents, respectively. For example, for state 1 in the positive PTA,

- *True Fluents = HighWater*
- *False Fluents = CriticalMethane, PumpOn*

Simultaneously, we populate another global field in the *Refiner* class, namely a matrix of states versus fluents, where boolean values are used to express which fluents hold at each state and which do not, as shown in Figure 6.7. Once again, the indices shown for states and fluents in the diagram coincide with the indices of the arrays storing the states and fluents, respectively, so that they can be used instead of the objects themselves. Easy access to the objects is possible by referring to their position in the relevant array using the indices.

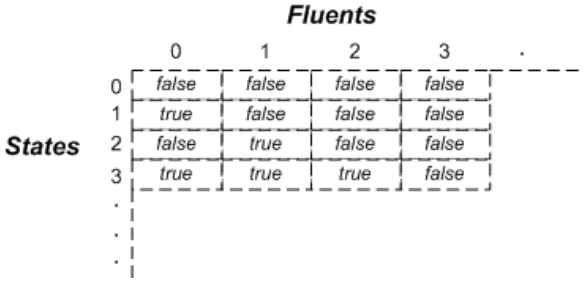

Figure 6.7: *States* × *Fluents* Matrix

Generalising the input positive scenarios

Following the construction of PTA trees that represent our input scenarios, the next step is to apply our state-merging algorithm on the PTA for the positive scenarios. The main structure used during this stage is shown in Figure 6.8, which gets populated as the states in the PTA are analysed and therefore aids the generalisation process. In addition, we prefer the states in the resulting model to have consecutive indices, hence we keep a counter which determines the index of an individual state in the PTA when it is added to the generalised model, or a state that has failed a merge with other states that share the same label.

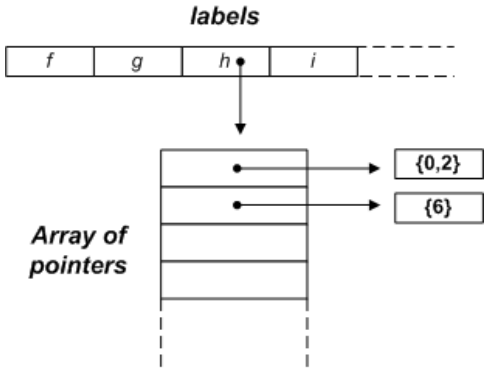

Figure 6.8: Internal Data Structure used for Generalisation Process

Each time a state in the positive PTA is analysed, its label is noted as a conjunction of (negations of) fluents according to the fluents that hold or not in that state. For instance, the label for state 1 in the positive PTA in which the only fluent with a true value is *HighWater*, is

$$\langle HighWater \wedge !PumpOn \wedge !Methane \rangle$$

If this label has not been encountered previously in any of the other states visited, then the label is added as a new element in the *labels* array shown in 6.8, and it is made to point at an ArrayList of ArrayLists which will classify all the states in the PTA which have this label.

Let us make this clearer using the running example of a positive PTA as shown in 6.9.

In Figure 6.10, *f, g, h, i* denote different labels encountered during the analysis of states in the positive PTA. Let us take the label h as an example, corresponding to $\langle !HighWater \wedge !PumpOn \wedge !Methane \rangle$. Suppose that during a particular phase of the generalisation process, the *groups* structure is as shown in Figure 6.10. Note that state 0 has been added to its appropriate list, and the remaining lists for fluents *f, g, i* have not been included due to space constraints.

Let us assume that state 7 is being visited. Its label is determined, which in this case is $\langle !HighWater \wedge !PumpOn \wedge !Methane \rangle$. We note that this label has already been visited, as it corresponds to label h. This means that there is potential for a merge to occur between states 0 and 7. We therefore use our testing code to check whether a merge between states 0 and 7 would be valid (according to the definition of *Merging* as outlined in Chapter 3), shown in Figure 6.11. As outlined in Chapter 4, the test passes if all traces

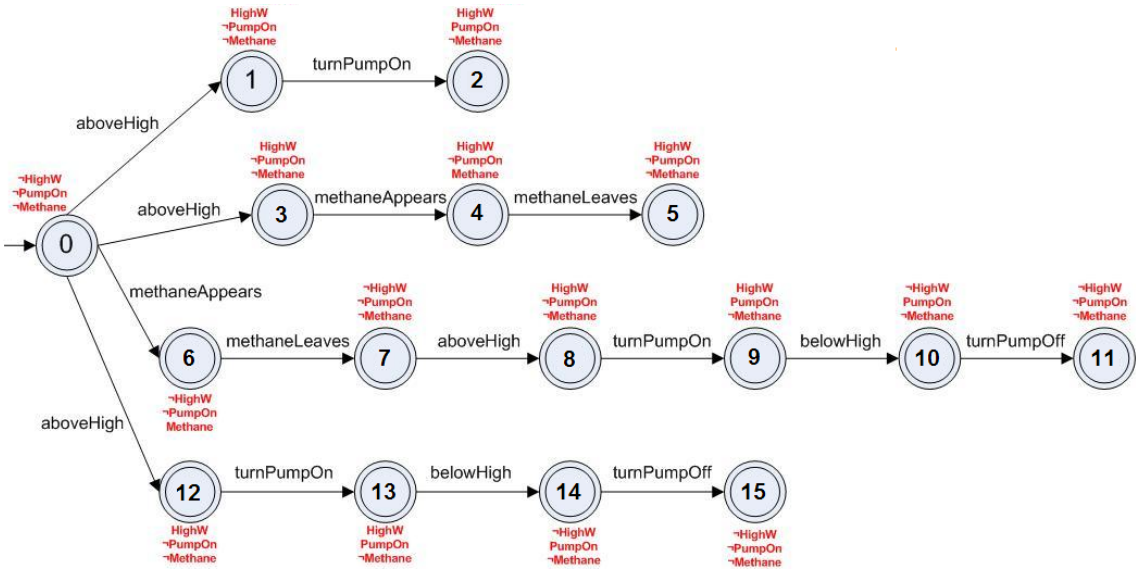

Figure 6.9: PTA for positive scenarios

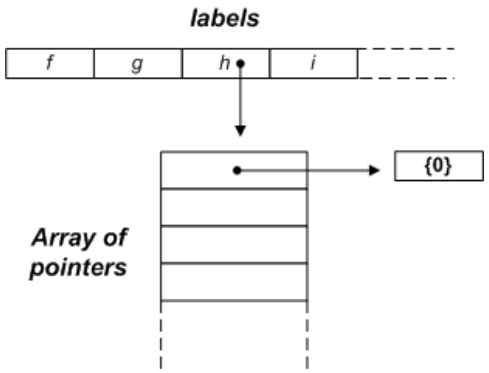

Figure 6.10: Initial *groups* Structure

resulting from the merge do *not* cover any of the negative scenarios. In this case, the traces that would be tested are those resulting from appending preceding traces of state 0 to succeeding traces of state 7, and by appending preceding traces of state 7 to succeeding traces of state 0.

However, state 0 has no preceding traces, so the test would check succeeding traces of 7. These are guaranteed to pass the test, since they are already present in the positive PTA and hence exhibit desirable behaviour.

If we assume that the given positive and negative scenarios are consistent, then *no* trace should appear in *both* the positive and the negative PTAs.

Examples of traces resulting by appending preceding traces of state 7 to succeeding traces of state 0 include:

- $\langle methaneAppears, methaneLeaves \rangle$ appended to $\langle aboveHigh, turnPumpOn \rangle$,
- $\langle methaneAppears, methaneLeaves \rangle$ appended to $\langle aboveHigh, methaneAppears, methaneLeaves \rangle$,

and so on. Our testing code implies the merge is valid, so the index of the new merged state is that of the state with the lowest index, in this case state 0. Hence, the *groups* structure gets updated to include just state 0 in the corresponding list, as shown in Figure 6.10.

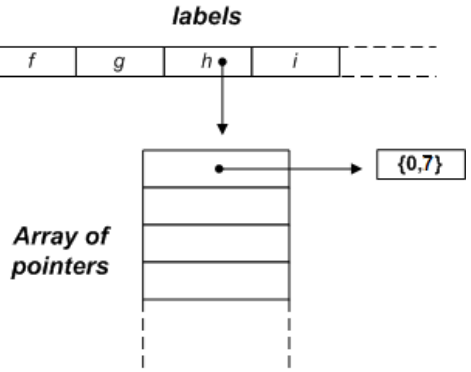

Figure 6.11: Updated *groups* Structure (i)

This also means that any predecessor and/or successor nodes of the original state 7 now need to become predecessors and/or successors of the new combined state 0 instead. The PTA is updated to reflect this, by iterating over the entire set of transitions in the model, and removing or adding transitions between states as required. In this case, state 7 has state 6 as a parent node, and state 8 as a child. Therefore, when the PTA is updated, the algorithm does the following:

- Adds state 8 as a child of state 0, and removes it from the children of state 7
- Adds state 6 as a parent of state 0, and removes it from the parents of state 7

- Replaces transitions (6, *methaneLeaves*, 7) and (7, *aboveHigh*, 8) with (6, *methaneLeaves*, 0) and (0, *aboveHigh*, 8), respectively

Now let us suppose that a potential merge between states leads to an invalid coalescence which incorrectly covers one or more of the negative scenarios. In this case the other states that share the same label as state 0, namely states 11 and 15, also pass the merging test.

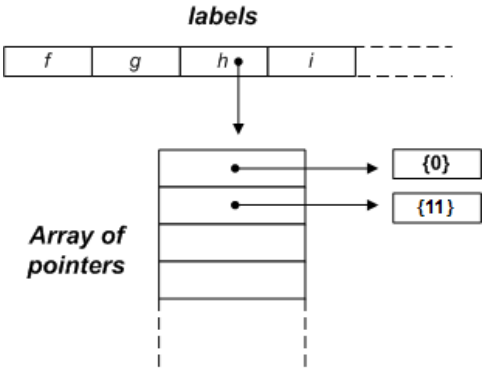

Figure 6.12: Updated *groups* Structure (ii)

However, to show the steps performed by our algorithm during an invalid merge, let us assume that state 11 fails the test. In this case, the state-merging algorithm would try and merge state 11 to any other states with the same label apart from state 0, but in this case there are none. Therefore a new list is created containing the index of state 11, and a new pointer is set up in the array of pointers as shown in Figure 6.12.

The remaining states are analysed and classified in the same way, and when all nodes in the PTA have been visited, then the resulting *groups* structure is used to populate fields related to the new structure, defined by the *GeneralPTA* class of the *refinement* package.

Refinement of initial LTS

Now that the generalised positive LTS is available, it can be used with the PTA corresponding to the negative scenarios in order to eliminate unwanted traces from the general system LTS.

The different refinement steps are conducted as explained in Chapter 3, by comparing transitions in the general system LTS to those in the generalised positive LTS and negative PTA. The *Fluent Conjunctions* method

(Chapter 3) is carried out by analysing the labels of the states involved, and consequently deciding whether to remove a trace in the general system LTS. When this happens, then the transition in question is removed from the set of transitions corresponding to the general LTS. Consequently, the predecessors and/or successors of the states involved are updated.

Once all the transitions have been considered and the appropriate decisions have been taken, there is a method that converts our resulting *GeneralPTA* object for the LTS into the LTSA representation so that it can be visualised in the LTSA window.

6.1.1 Discussion

No refinement when no negative scenarios available

Note that in the code listing provided in Listing 6.1, line 30 makes sure that the refinement procedure only takes place when there are negative scenarios available. This is because the main aim of the refinement is to convert an existing general LTS into a more specific one which does *not* allow any of the negative behaviour portrayed through the set of negative scenarios, whilst simultaneously preserving the desirable traces.

Performing the refinement process in the absence of any examples of negative behaviour may just eliminate traces that are in fact *not* undesirable, thus we would be over-constraining the model.

Updating states following elimination of a transition

It is important to realise that due to the deterministic nature of the LTSs that we assume to be dealing with, it is not possible to have multiple transitions between a pair of states. Therefore, when a transition is removed from the system LTS as a result of the refinement procedure, then this implies that the pair of nodes which were connected by this transition are no longer connected. Hence, for each state, the set of predecessor or successor nodes need to be updated to exclude the state which is no longer a parent or a child of the state in question. For instance, the elimination of a Transition t from state a to state b will result in the removal of state a as a parent of state b, and the removal of state b as a child of state a.

6.2 User Interface

The interaction between the new *refinement* package that we have created and the current *UI* package of the LTSA tool is shown in Figure 6.13 and

explained in this section.

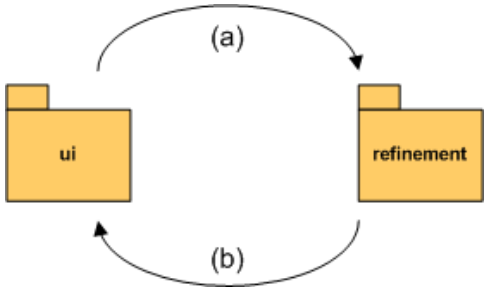

Figure 6.13: Interaction between *UI* and new *refinement* package

The *Refinement* Feature

In order to integrate our implementation with the existing LTSA source code, we have added a *RefinedAction* class that implements the *ActionListener* interface in the *UI* package. This class contains an *actionPerformed* method, which is responsible for invoking the appropriate methods in the *refinement* package during the refinement process. The arrow labelled *(a)* in Figure 6.13 illustrates this relation, and the corresponding code for the *actionPerformed* method is shown in Listing 6.1. The arrow labelled *(b)* in Figure 6.13 corresponds to everything that is returned by our algorithm as a result. This includes the representation of the positive and negative scenarios using PTAs, the generalised model for the positive PTA, and the refined model for the system.

Firstly, an exact copy of the general system is made (line 4), and then it is converted into our internal representation of LTSs (line 6). This enables us to execute the code in the *refinement* package on the converted model, and as a result perform the various stages of our refinement algorithm. Amongst other operations, the *convert* method enables the information extracted by the existing parser code in the LTSA tool to be used to initialise some of the global fields in our *Refiner* class. This includes the set of events for the input system, as well as the different fluents and their corresponding definitions.

The calls to the *readNegScenarios* and *readPosScenarios* methods from the *Refiner* class at lines 15 and 21, respectively, lead to the representation of user-provided scenarios as PTAs. Once all the relevant information has been taken into account and the trees corresponding to the positive and negative scenarios are fully populated, the calls to *makeCompactState* on

Listing 6.1: Code listing for the *RefineAction* class in UI

```
class RefineAction implements ActionListener {
  public void actionPerformed(ActionEvent e) {

  CompactState system = (CompactState)current
                        .composition.myclone();
  GeneralPTA sysLTS = Refiner.convert(system);
  GeneralPTA negPTA = new GeneralPTA();
  GeneralPTA posPTA = new GeneralPTA();
  GeneralPTA genpos = new GeneralPTA();
  CompactState negLTS = new CompactState();
  CompactState posLTS = new CompactState();
  CompactState genPosLTS = new CompactState();

  if(!negText.equals(neg_scen.getText())){
    negPTA = Refiner.readNegScenarios(negScenarios);
    negLTS = Refiner.makeCompactState("Negative PTA",
                                      negPTA);
    current.machines.add(negLTS);
  }
  if(!posText.equals(pos_scen.getText())){
    posPTA = Refiner.readPosScenarios(posScenarios);
    posLTS = Refiner.makeCompactState("Positive PTA",
                                      posPTA);
    current.machines.add(posLTS);
    genpos = Refiner.synthesisePositive();
    genPosLTS = Refiner.makeCompactState
                ("Generalised Positive LTS",genpos);
    current.machines.add(genPosLTS);
  }
  if(!negText.equals(neg_scen.getText())){
    GeneralPTA result = Refiner.getInstance()
                        .initRefine(negPTA,genpos,sysLTS);
    CompactState finalLTS = Refiner.makeCompactState
                            ("Refined Result",result);
    current.machines.add(finalLTS);
  }
  postState(current);
  }
}
```

lines 16 and 22 are used to convert our internal representation of the PTA trees to structures that are used by the existing LTSA source code. These are added to the list of diagrams to be displayed, by executing the code at lines 18 and 24.

Following this, the *synthesisePositive* method call at line 25 performs the coalescence of states in the positive PTA. The result, which uses our internal representation of the corresponding structure, is once again converted to the LTSA representation at line 26 and added to the list of diagrams at line 28.

Now that the generalised positive LTS is available, it can be used together with the negative PTA and the given LTS in order to refine the given model. Note that line 30 ensures that refinement is only carried out if the user has provided any negative scenarios, otherwise it would not be possible to perform the refinement using only the given positive scenarios.

Thereafter, the method called at line 31 is one of the refinement methods in our *Refiner* class. This makes calls to other private refinement methods which deal with subsequent stages of the refinement procedure. The result is converted to the LTSA representation at line 33, and then added to the list of diagrams to be displayed (line 35). Finally, line 37 enables the visualisation of all the required diagrams.

Additions to the LTSA GUI

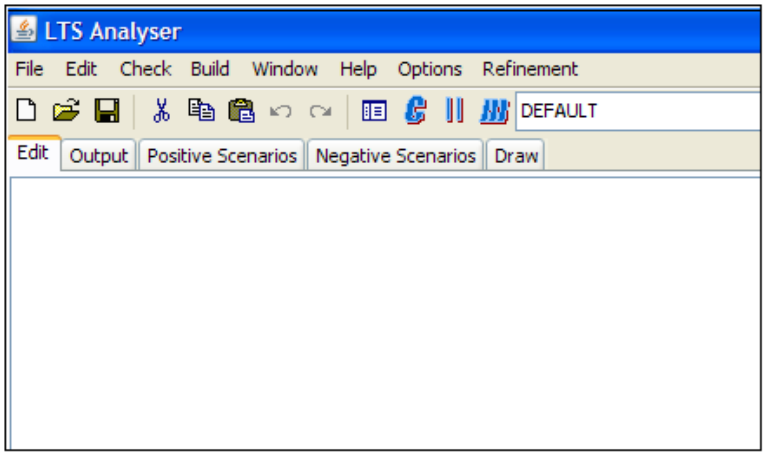

Figure 6.14: Additional features in LTSA interface

Upon launching the LTSA tool, users can see the window shown in Figure 6.14, which is the standard LTSA tool but with a few additions from

our part. The first new functionality we encounter is the new menu item *Refinement*. This is a *JMenu* object that is added to the existing *JMenuBar* including the items *File*, *Edit*, *Check*, *Build*, and so on.

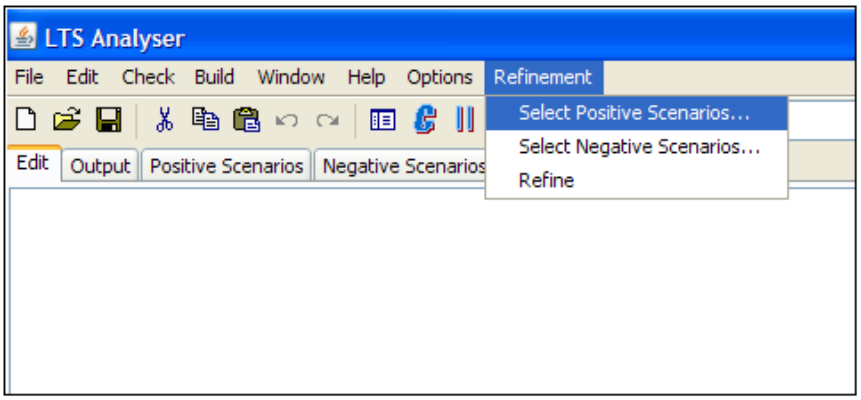

Figure 6.15: Options in the new menu item

In addition, three *JMenuItem* objects are added under *Refinement*, as shown in Figure 6.15. The first two are used to open the files containing positive and negative scenarios, respectively, similarly to the way in which *File* → *Open* helps us open an FSP specification whose resulting model we want to refine in this case.

The *Select Positive Scenarios* and *Select Negative Scenarios* options work exactly the same way in which FSP files are opened using *File* ⟶ *Open*. In our case, *Refinement* ⟶ *Select Positive Scenarios* and *Refinement* ⟶ *Select Negative Scenarios* enable the File Dialog windows to appear and hence allow the user to select the appropriate files that contain the relevant scenarios.

As you can see, there are two additional tabs other than the *Edit*, *Output* and *Draw* tabs, namely *Positive Scenarios* and *Negative Scenarios*. These are two new *JEditorPane* objects added to the existing *JTabbedPane*. When the relevant files are opened, their contents are displayed in the new tabs using a buffer, as shown in Figures 6.16 and 6.17.

Note that a we could have used a single tab to contain *all* scenarios, but the reason for not doing so is due to user-friendliness, particularly if the tool is extended later on to enable dynamic generation of scenarios. Users can then add new scenarios under the appropriate tabs.

Once we have our scenarios and FSP specification ready, we can use the third option of the *Refinement* menu item this time, named *Refine* (Figure

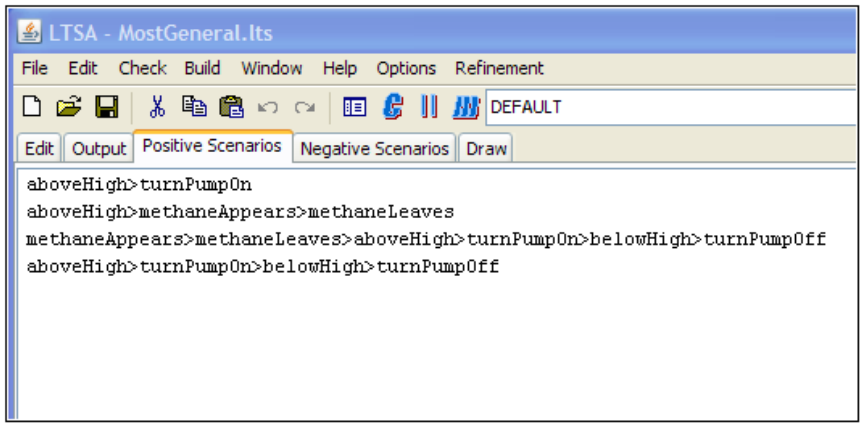

Figure 6.16: Positive Scenarios file contents

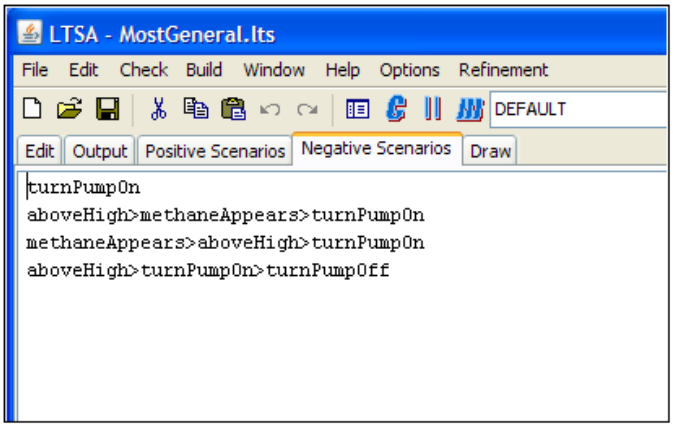

Figure 6.17: Negative Scenarios file contents

6.18). This is responsible for invoking the relevant *ActionListener* that initiates the refinement process and executes the following steps:

1. The given positive and negative scenarios are read in their textual forms displayed in their respective tabs, and corresponding diagrams are produced which enable the user to view these scenarios diagrammatically (Figures 6.19 and 6.20). These are named *Negative PTA* and *Positive PTA*, respectively.

2. The positive PTA in Figure 6.19 is generalised using our State-Merging algorithm, and can be viewed under the *Draw* tab (Figure 6.22 as the

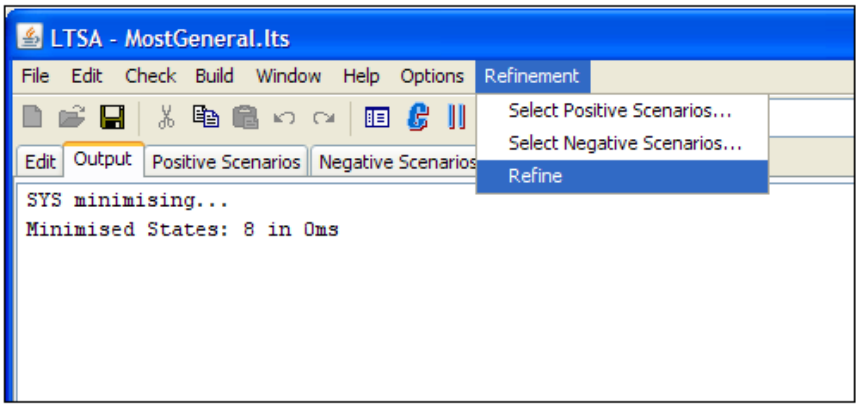

Figure 6.18: The "refine" option

Generalised Positive LTS.

3. The PTA structure for the negative scenarios and the generalised model for the positive scenarios are used in conjunction with the given system model (Figure 6.21) to perform the actual refinement as specified in Chapter 3.

4. An LTS representation of the resulting model is finally created and made available to the user for viewing under the *Draw* tab (Figure 6.23, as the *Refined Result*.

The next chapter shows the application of our refinement algorithm described in this report on a larger system, and the results obtained are then compared to the output of the ILP approach on the same example.

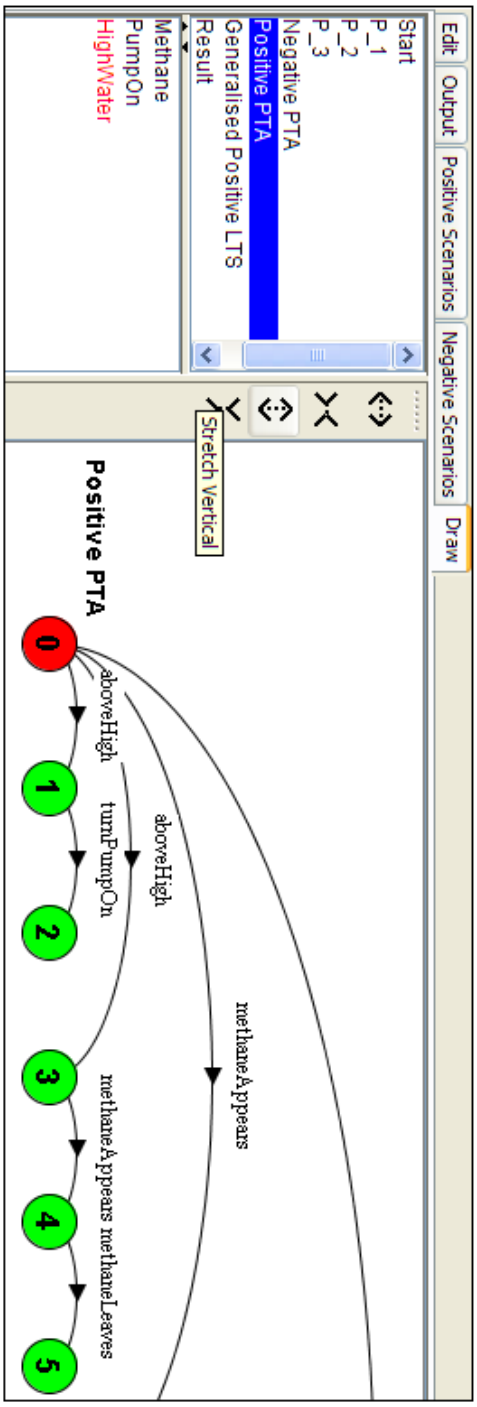

Figure 6.19: Diagram representing positive scenarios

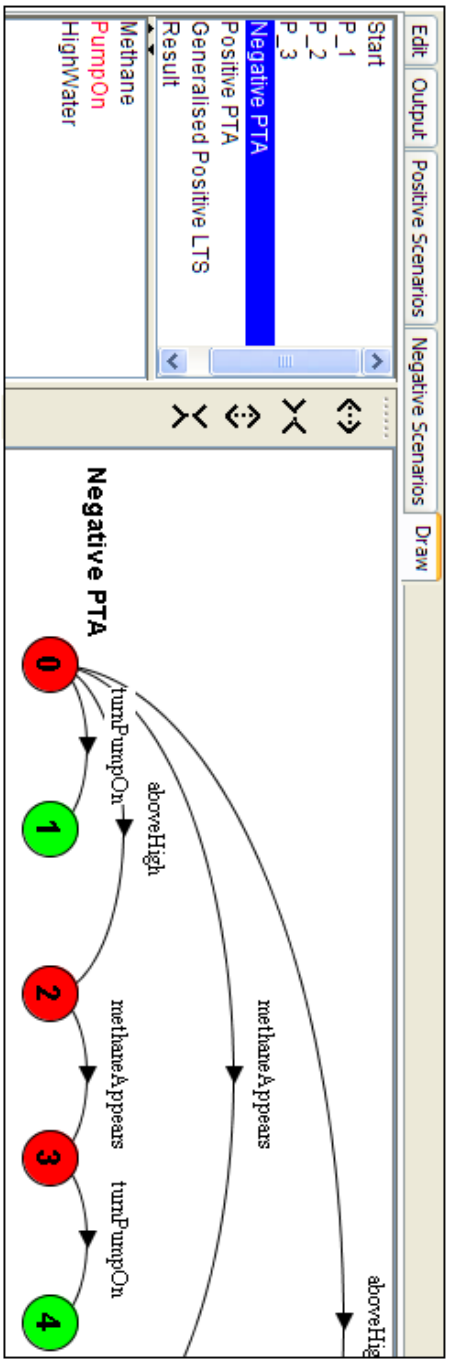

Figure 6.20: Diagram representing negative scenarios

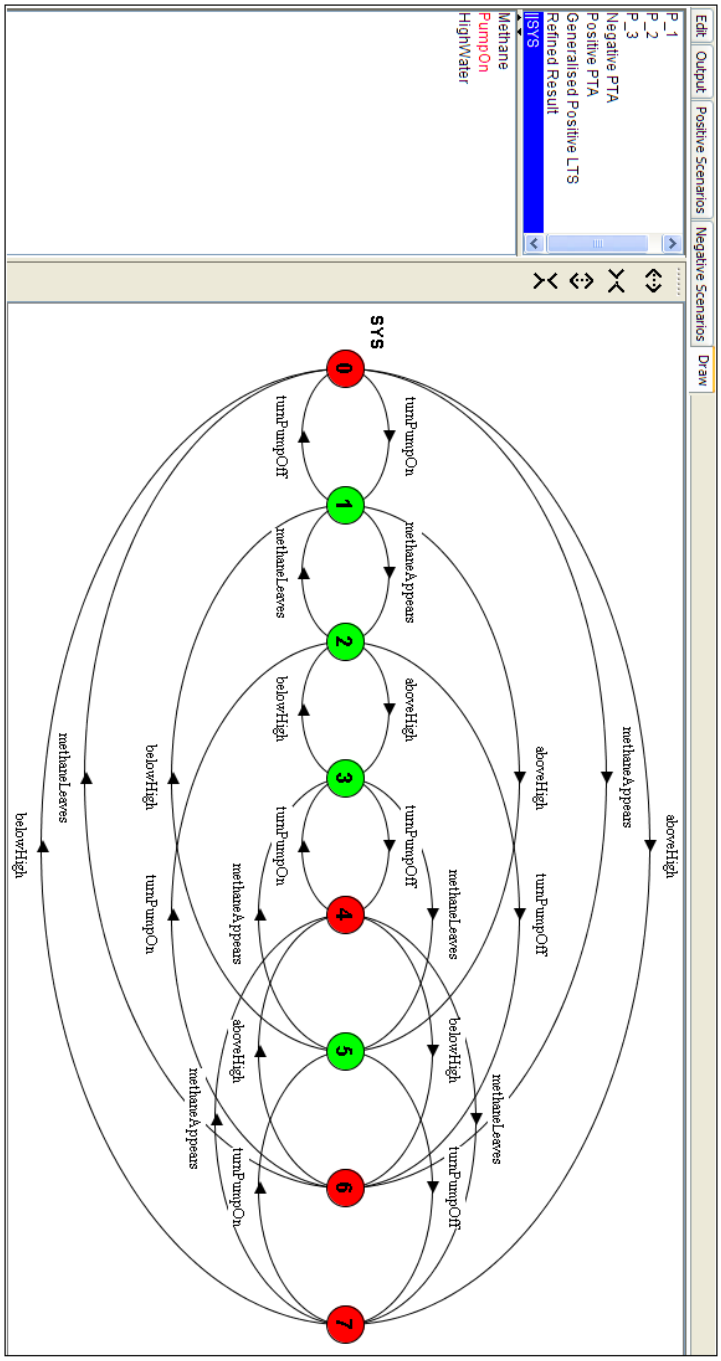

Figure 6.21: Diagram representing initial system model

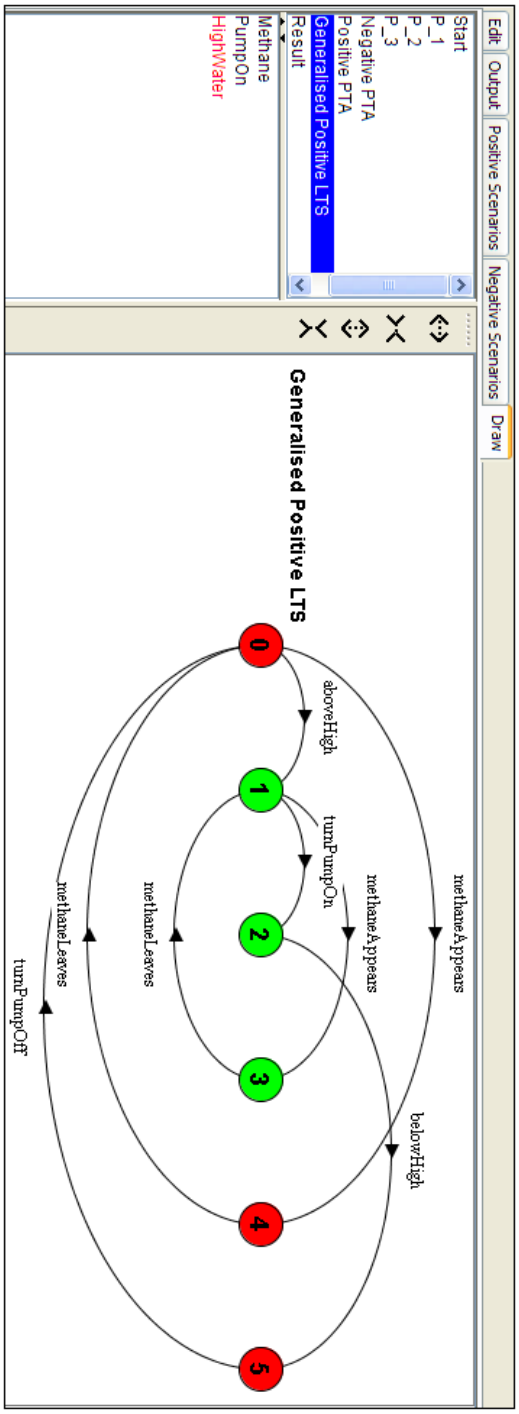

Figure 6.22: Diagram representing generalised positive behaviour

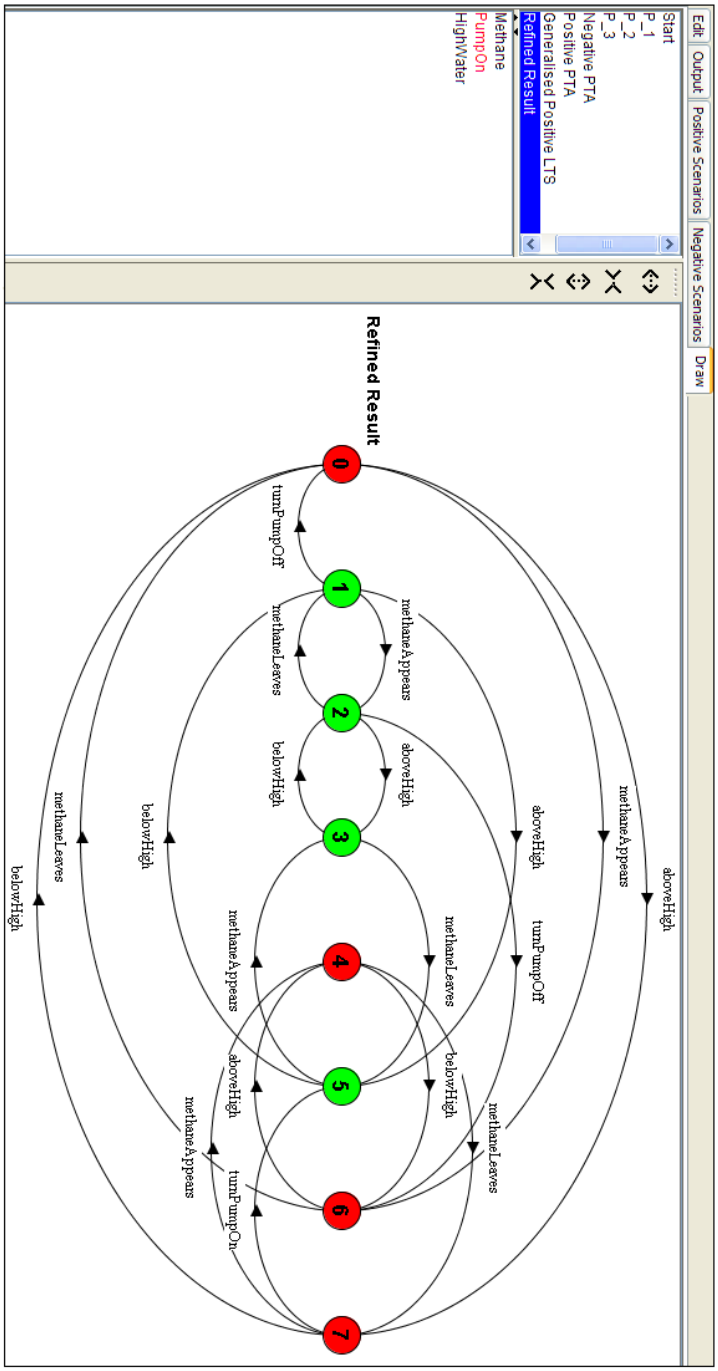

Figure 6.23: Diagram representing refined system model

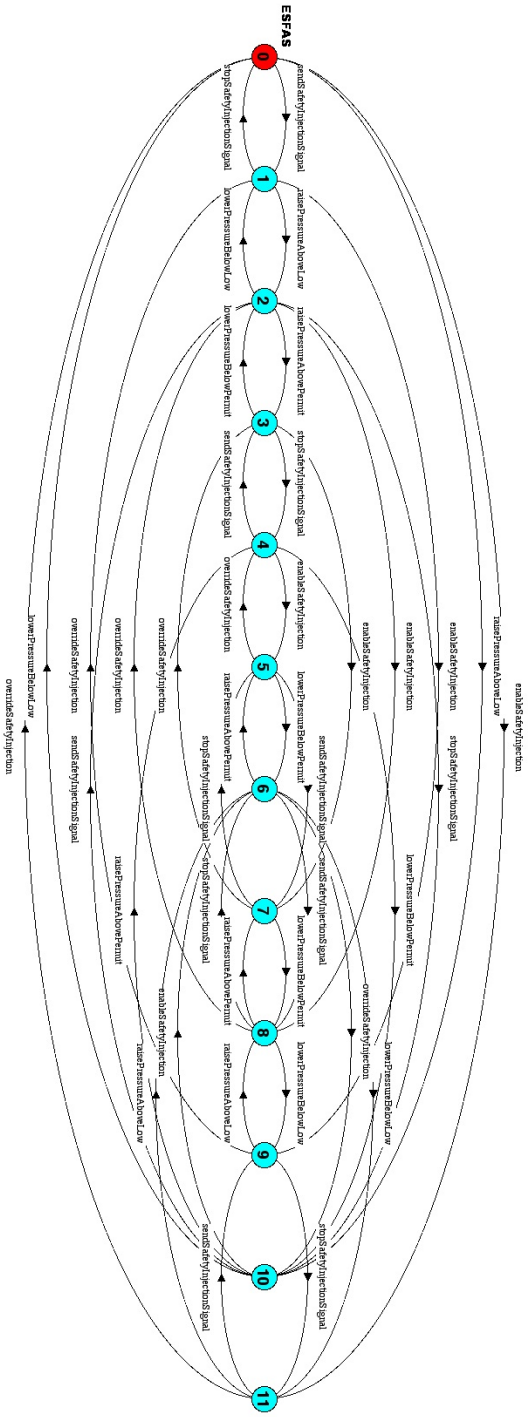

Figure 6.24: General System LTS for ESFAS

Chapter 7

Testing: A Case Study

In this chapter we have provided a complete working example of our algorithm on a sample case study in [7, 22]. The system in question is *The Engineered Safety Feature Actuation System (ESFAS)* of a nuclear power plant, which prevents or mitigates damage to the core and coolant system on the occurrence of a fault such as a loss of coolant. The ESFAS monitors the water pressure of the coolant system. If the pressure falls below some predetermined setpoint, the system sends a safety injection signal to the safety feature components. the function of which is to cope with the accident. A manual block (pushbutton) is provided in order to override a safety injection signal and to avoid actuation of the protection system during a normal start-up or cool-down phase. A manual block is permitted if and only if the steam pressure is below a specified value, and is effective if and only if it is executed before the protection signal is sent. The manual block must be automatically reset by the system.

We can open the FSP file corresponding to this system using the LTSA tool. The process is compiled and composed using the appropriate LTSA options, and the corresponding diagrammatic representation of the ESFAS system can be seen in Figure 6.24.

The corresponding set of events, also called the alphabet A for this system is:

$A = \{$*overrideSafetyInjection, enableSafetyInjection, sendSafetyInjectionSignal, stopSafetyInjectionSignal, raisePressureAbovePermit, raisePressureAboveLow, lowerPressureBelowPermit, lowerPressureBelowLow*$\}$

The corresponding set of fluents F for this system is:
$F = \{$*SafetyInjection, Overridden, PressureBelowLow, PressureAbovePermit*$\}$

Due to the large size of the LTS in Figure 6.24, it has not been possible to include information regarding the fluents that hold at each of its states. Therefore, we have presented this information in Table 7.1.

State	**SafetyInjection**	**Overridden**	**Pressure BelowLow**	**Pressure AbovePermit**
0	×	✓	✓	×
1	✓	✓	✓	×
2	✓	✓	×	×
3	✓	✓	×	✓
4	×	✓	×	✓
5	×	×	×	✓
6	×	×	×	×
7	✓	×	×	✓
8	✓	×	×	×
9	✓	×	✓	×
10	×	✓	×	×
11	×	×	✓	×

Table 7.1: Fluent values at States in General LTS

We are going to consider just one positive scenario for this example, namely:

- $\langle raisePressureAboveLow, raisePressureAbovePermit, enableSafetyInjection, lowerPressureBelowPermit, lowerPressureBelowLow, sendSafetyInjectionSignal\rangle$

and the following negative scenarios,

- $\langle$*sendSafetyInjectionSignal*$\rangle$
- $\langle$*raisePressureAboveLow, enableSafetyInjection*$\rangle$
- $\langle$*raisePressureAboveLow, sendSafetyInjectionSignal*$\rangle$
- $\langle$*enableSafetyInjection*$\rangle$

These are stored in separate files, which are opened using *Refinement* $\longrightarrow$ *Select Positive Scenarios...*, to show the chosen positive scenarios in the *Positive Scenarios* tab and similarly *Refinement* $\longrightarrow$ *Select Negative Scenarios...* populates the *Negative Scenarios* tab with the chosen negative scenarios.

Algorithm 1 is used on both sets of scenarios to construct their respective PTAs. In the case of the positive scenario, a tree with state 0 as the root is created according to line 3 in the algorithm, and the *for*-loop at line 5 is entered once since we only have one example of positive behaviour. The root node is then set to be the starting point, and the inner *for*-loop at line 7 is executed.

Each time an event label is read from the positive trace, a new state is created with the current index (line 8) which is added to the tree as a child of the starting node (line 9). Similarly, the starting node is set to be a parent of the new node (line 10), and the entire transition is stored as a 3-tuple in the list of transitions for the resulting PTA (line 11).

The new node is now set to be the starting node (line 12), the index of the next state is incremented (line 13) as we have just created a node with the previous index, and the inner loop is repeated as many times as the number of transitions in the positive trace.

Note that each time a state is created, its private field *accepting* is set to *true*, since all states in a positive PTA can be accepting (Chapter 4). The resulting positive PTA is shown in Figure 7.1.

In the case of the negative scenarios, there are more of them compared to the single positive scenario, and so the number of steps required is slightly greater this time. Lines 3 and 4 of Algorithm 1 are executed as before, and the outer *for*-loop is executed. The first scenario is read, and the appropriate state (with index 1) and transition (with event label *sendSafetyInjectionSignal*) are added to the negative PTA. The inner *for*-loop is executed just once for the first scenario, since only one transition is available in this case.

However, when we have finished reading the first scenario, the index of the next state to be created is set to 2, and the outer *for*-loop is executed for the next negative trace. Note that this time, only the final states have their *accepting* field set to true, so each time the algorithm finishes reading an entire scenario, it sets the state at *[current index -1]* to be *accepting* (since the current index contains the number for the *next* state to be created).

The corresponding negative PTA can be seen in Figure 7.2.

As explained in previous chapters, the first step is the generalisation of positive system behaviour. States in the positive PTA are visited in

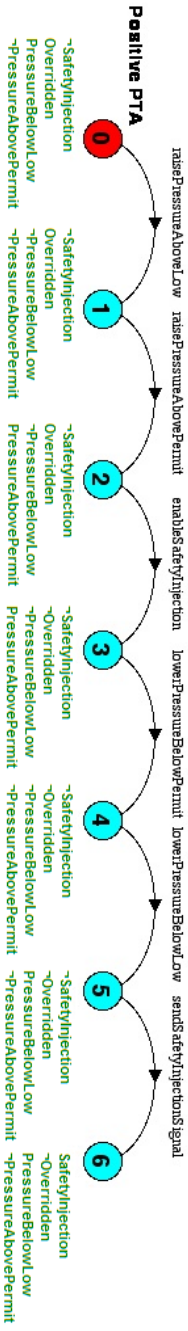

Figure 7.1: PTA for Positive Scenarios

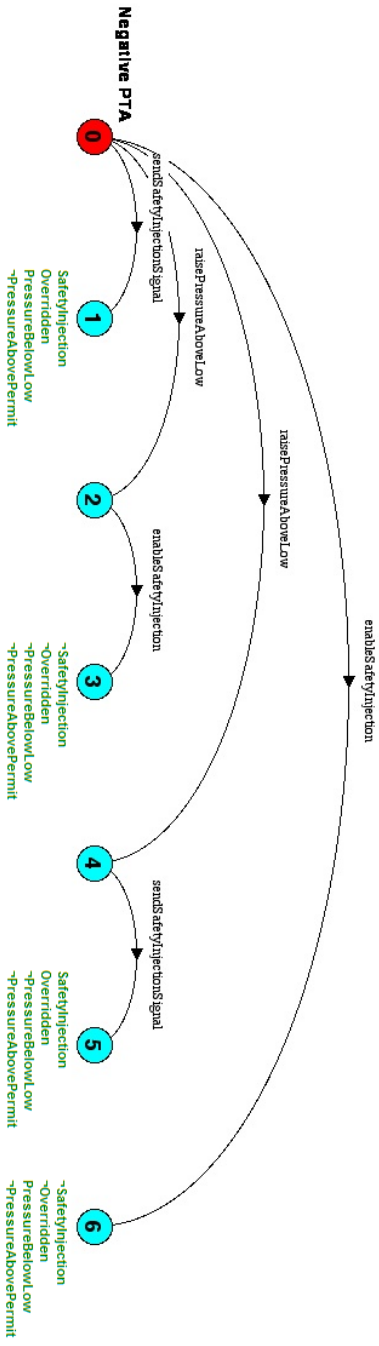

Figure 7.2: PTA for Negative Scenario

increasing order of state index. We remind ourselves that the numbering of states in a PTA is performed *along* the branches in a depth-first fashion, such that *all* states in a particular branch are visited before traversing other branches. In our case, we only have *one* positive scenario, and Figure 7.1 shows that each state in the positive PTA is defined by a different set of fluent values. Hence there is no scope for any merge. Applying our state-merging algorithm in Algorithm 2 to the positive PTA would leave it unchanged.

Following this, our refinement algorithm is applied to the general system LTS shown in Figure 6.24, taking into account the negative PTA and the positive generalised LTS (which in this case is our positive PTA). Let us explain each step of the refinement process in more detail. We start off by executing the code in Algorithm 4.

Firstly, we initialise the list of *victims*, which will subsequently get populated with tuples *(e,n)*. Each tuple represents an event e outgoing from a state n in the given LTS, where the transition in question has already been removed from the initial LTS. Initially the *victims* list is empty (line 4 in Algorithm 4).

We now look at the accepting states in the negative PTA, namely the final ones, since the incoming transitions into these states mark the undesirability of these traces. In this case the final states are 1, 3, 5 and 6. Each of these is studied in turn according to the *for*-loop at line 5.

Let us look at state 1 in the negative PTA. At line 6, we look at its incoming transition labelled *sendSafetyInjectionSignal* which is undesirable and should therefore be removed from the given LTS. We therefore note the fluent values that hold at state 1:

$\langle$ *SafetyInjection, Overridden, PressureBelowLow,* $\neg$*PressureAbovePermit*$\rangle$,

and we use these to find equivalent states in the given LTS which have an incoming transition labelled *sendSafetyInjectionSignal* (line 7).

Looking closely at Figure 6.24, we can see that state 1 in the given LTS satisfies this condition. According to the *for*-loop at line 8, we need to make note of the states in the LTS that have the undesirable transitions outgoing from them (line 9), and in this case state 1 in the given LTS is receiving the undesirable transition from state 0. Consequently, this information is added to *victims* (line 10) in the form of a tuple, (*sendSafetyInjectionSignal*, 0), and the *sendSafetyInjectionSignal* transition into state 1 is thus removed from the LTS.

We now look at state 3 in the negative PTA. At line 6, we look at its incoming transition labelled *enableSafetyInjection* which is undesirable and

should therefore be removed from the given LTS. We therefore note the fluents that hold at state 3 and use them to find equivalent states in the given LTS which have an incoming transition labelled *enableSafetyInjection* (line 7). In this case the fluent values at state 3 are

$$\langle \neg SafetyInjection, \neg Overridden, \neg PressureBelowLow, \neg PressureAbovePermit \rangle$$

We can see that state 6 in Figure 6.24 shares these fluent values with state 3 in the negative PTA. According to the *for*-loop at line 8, we make note of the states in the LTS that have the undesirable transitions outgoing from them (line 9), which in this case is state 10 in the general LTS.

Consequently, this information is added to *victims* (line 10) as (*enableSafetyInjection*, 10), and the *enableSafetyInjection* transition into state 6 is removed from the LTS.

The steps described above are repeated for the remaining final states in the negative PTA, but for different events. In the case of state 5, an incoming transition labelled *sendSafetyInjectionSignal* is undesirable. The corresponding state in the given LTS is state 2, which has an incoming *sendSafetyInjectionSignal* from state 10. This results in the elimination of the transition from state 10 to state 2 (line 11) in the given LTS, and the addition of (*sendSafetyInjectionSignal*, 10) to the *victims* list, according to line 10 in the algorithm.

As for state 6, it is not possible for it to accept an incoming transition labelled *enableSafetyInjection*, and its equivalent state in the given LTS is state 11, with an incoming transition labelled *enableSafetyInjectionSignal* from state 0. This transition is removed from the given LTS (line 11), and (*enableSafetyInjectionSignal*, 0) is added to the *victims* list (line 10).

Having looked at each of the final states in the negative PTA, our *victims* set which contains the following elements is given as a parameter to the method described in Algorithm 5, as set out in line 14 of Algorithm 4. Table 7.2 shows the transitions removed so far, and as we remove further transitions from the general LTS, we will update this table in order to facilitate the analysis and verification of the final results.

$$\boldsymbol{victims} = \{(sendSafetyInjectionSignal, 0), (enableSafetyInjection, 10), (sendSafetyInjectionSignal, 10), (enableSafetyInjectionSignal, 0)\}$$

We now analyse each remaining transition in the given LTS according to the *for*-loop at line 5 of Algorithm 5, in conjunction with the tuples in *victims*.

Transitions removed - *(source state, event, target state)*
(0, *sendSafetyInjectionSignal*, 1)
(10, *enableSafetyInjection*, 6)
(10, *sendSafetyInjectionSignal*, 2)
(0, *enableSafetyInjection*, 11)

Table 7.2: Fluent values at States in General LTS

Transitions are not considered in any particular order, instead an iterator is run across the set of transitions for the system, and each transition is handled separately. Note that only those transitions which are labelled with one of the event labels in {*sendSafetyInjectionSignal, enableSafetyInjection*} will be analysed, as we can see on line 6 in Algorithm 5. This is because only these event labels appear as the undesirable ones amongst the given negative scenarios. The transitions that thus satisfy the *if*-condition at line 6 for each undesirable event label are shown in Table 7.3 in the form *(source state, event label, target state)*.

Transitions with label e = ***sendSafetyInjectionSignal***	**Transitions with label** f = ***enableSafetyInjection***
(4, e, 3)	(4, f, 5)
(5, e, 7)	(1, f, 9)
(6, e, 8)	(2, f, 8)
(11, e, 9)	(3, f, 7)

Table 7.3: Transitions to analyse in given LTS

Let us look at the first *sendSafetyInjectionSignal* transition. For transition (4, *sendSafetyInjectionSignal*, 3), the algorithm firstly checks if there is a transition in the generalised positive LTS with the same event label (line 7). As seen in Figure 7.1, the transition from state 5 to state 6 satisfies this condition. In lines 8-10, the fluent values of state 4 in the general LTS and state 5 in the positive LTS are considered.

State 4 in the general LTS is defined by the fluent values
$\langle \neg SafetyInjection,\ Overridden,\ \neg PressureBelowLow,\ PressureAbovePermit \rangle$

On the other hand, state 5 in the positive LTS is defined by the fluent values

$$\langle \neg SafetyInjection,\ \neg Overridden,\ PressureBelowLow,\ \neg PressureAbovePermit \rangle$$

As we can see, the fluent values are *not* exactly the same, therefore the *if*-condition on line 10 is satisfied, and the method that further analyses these states according to their fluents is called at line 11.

Algorithm 6 outlines the next steps involved. The method takes in as input the state in the given LTS whose outgoing transition is under analysis, i.e. state 4 in this case, together with the *victims* list (line 1). State 4 is then compared to other states from the given LTS whose outgoing *sendSafetyInjectionSignal* transitions have already been removed and are contained in the *victims* list (line 5). In this case we can see that the *victims* list has two entries that satisfy this:

$$\{(sendSafetyInjectionSignal, 0), (sendSafetyInjectionSignal, 10)\}$$

The first element refers to the transition from state 0 to state 1 which was previously removed, and the second one is the former transition from state 10 to state 2. We compare states 4 and 0 first.

As mentioned before, state 4 is defined by the fluent values

$$\langle \neg SafetyInjection,\ Overridden,\ \neg PressureBelowLow,\ PressureAbovePermit \rangle$$

whilst state 0 is defined by

$$\langle \neg SafetyInjection,\ Overridden,\ PressureBelowLow,\ \neg PressureAbovePermit \rangle$$

At line 6, the intersection *inter* of the fluent values for both states is determined:

$$\boldsymbol{inter} = \{\neg SafetyInjection,\ Overridden\}$$

According to line 7, we construct the power set *power* of elements in *inter*, in increasing order of size and excluding the empty set:

$$\boldsymbol{power} = \{\{\neg SafetyInjection\}, \{Overridden\}, \{\neg SafetyInjection,\ Overridden\}\}$$

At line 8 we consider the first *subset*, namely $\{\neg SafetyInjection\}$. We need to check if this fluent value occurs in any of the states in the positive LTS with an outgoing transition labelled *sendSafetyInjectionSignal*. As mentioned before, state 5 in the positive LTS has an outgoing transition labelled *sendSafetyInjectionSignal*, and is defined by the fluents

$$\langle \neg SafetyInjection,\ \neg Overridden,\ PressureBelowLow,\ \neg PressureAbovePermit \rangle$$

and so the *if*-condition at line 9 in Algorithm 6 is satisfied, and another subset from *power* is fetched according to line 10 and then line 8.

We now look at *subset* $\{Overridden\}$. Checking the fluent values of state 5 once again, we realised that this fluent value is not amongst them, and so this time the *if*-condition at line 9 is not satisfied, so the *else*-case on line 11 is executed. According to line 12, the *sendSafetyInjectionSignal* transition outgoing from state 4 in the general LTS is eliminated, and the procedure is exited by returning the modified general LTS, following the execution of lines 13, 16, 17, and finally 22. The table including all the removed transitions is therefore updated, as shown in Table 7.4

Transitions removed - *(source state, event, target state)*
(0, *sendSafetyInjectionSignal*, 1)
(10, *enableSafetyInjection*, 6)
(10, *sendSafetyInjectionSignal*, 2)
(0, *enableSafetyInjection*, 11)
(4, *sendSafetyInjectionSignal*, 3)

Table 7.4: Transitions removed from General LTS (i)

We then go back to line 11 where the calling method in Algorithm 5 called Algorithm 6, and exit the *if*-condition at line 12. Lines 15 and 16 are executed subsequently, and we are back at line 5, where we consider the next transition in the general LTS.

Let us suppose the next transition that meets the *if*-condition at line 6 is (5,*sendSafetyInjectionSignal*, 7). Once again we look for a state in the positive LTS which has the same outgoing transition. As mentioned in the previous example, this corresponds to state 5 in the positive LTS, which is defined by the fluents

$$\langle \neg SafetyInjection,\ \neg Overridden,\ PressureBelowLow,\ \neg PressureAbovePermit \rangle$$

This is compared to the fluent values holding at state 5 in the general LTS, which are

$$\langle \neg SafetyInjection,\ \neg Overridden,\ \neg PressureBelowLow,\ PressureAbovePermit \rangle$$

Similarly to the previous example, their fluent values do *not* match, so we need to call Algorithm 6 again at line 11.

As before, the relevant entries in the *victims* list that we need to use are:

$$\{(sendSafetyInjectionSignal,0),\ (sendSafetyInjectionSignal,10)\}$$

This time let us look at the second element, namely (*sendSafetyInjectionSignal*, 10). State 10 is defined by the fluent values

$$\langle\ \neg SafetyInjection,\ Overridden,\ \neg PressureBelowLow,\ \neg PressureAbovePermit\rangle$$

We construct the intersection of fluents that occur in states 5 and 10 of the given LTS (line 6), and the corresponding power set (line 7):

$$\boldsymbol{inter} = \{\neg SafetyInjection,\ \neg PressureBelowLow\}$$

$$\boldsymbol{power} = \{\{\neg SafetyInjection\},\ \{\neg PressureBelowLow\},\ \{\neg SafetyInjection,\ \neg PressureBelowLow\}\}$$

From the previous example, we know that $\{\neg SafetyInjection\}$ occurs in state 5 in the positive LTS, hence the *if*-condition on line 9 is met, and line 10 is executed. We therefore choose the next subset in *power*, namely $\{\neg PressureBelowLow\}$. This time, state 5 in the positive LTS does *not* include this fluent value, so the *else*-case at line 11 is executed, leading to the elimination of transition (5, *sendSafetyInjectionSignal*, 7) from the system LTS (line 12). Following execution of lines 13, 16, and 17, the modified system LTS is returned at line 22 and we resume execution of Algorithm 5. The table containing removed transitions is once again updated, as shown in Table 7.5

Transitions removed - *(source state, event, target state)*
(0, *sendSafetyInjectionSignal*, 1)
(10, *enableSafetyInjection*, 6)
(10, *sendSafetyInjectionSignal*, 2)
(0, *enableSafetyInjection*, 11)
(4, *sendSafetyInjectionSignal*, 3)
(5, *sendSafetyInjectionSignal*, 7)

Table 7.5: Transitions removed from General LTS (ii)

The *for*-loop at line 5 in Algorithm 5 continues to execute on the remaining transitions in the system LTS, and as we mentioned previously, only those in Table 7.3 are analysed. As a result of similar steps as those explained in the two examples outlined above, transitions (6, *sendSafetyInjectionSignal*, 8), (1, *enableSafetyInjection*, 9), (2, *enableSafetyInjection*, 8), and (3, *enableSafetyInjection*, 7) are also removed from the given LTS. Table 7.6 reflects this.

Transitions removed - *(source state, event, target state)*
(0, *sendSafetyInjectionSignal*, 1)
(10, *enableSafetyInjection*, 6)
(10, *sendSafetyInjectionSignal*, 2)
(0, *enableSafetyInjection*, 11)
(4, *sendSafetyInjectionSignal*, 3)
(5, *sendSafetyInjectionSignal*, 7)
(6, *sendSafetyInjectionSignal*, 8)
(1, *enableSafetyInjection*, 9)
(2, *enableSafetyInjection*, 8)
(3, *enableSafetyInjection*, 7)

Table 7.6: Transitions removed from General LTS (iii)

In the case of transition (11, *sendSafetyInjectionSignal*, 9) the *if*-condition at line 7 of Algorithm 5 is entered, as we know there exists a transition with the same event from state 5 to state 6 in the positive LTS. State 11 in the system LTS is defined by the fluents

$$\langle \neg SafetyInjection,\ \neg Overridden,\ PressureBelowLow,\ \neg PressureAbovePermit \rangle$$

which coincide exactly with the fluent values for state 5 in the positive LTS. This causes the *if*-condition at line 10 to *not* be met, and so no further analysis is performed on the transition in question. Hence, the outgoing transition labelled *sendSafetyInjectionSignal* between states 11 and 9 remains in the system LTS.

A similar procedure occurs with transition (4, *enableSafetyInjection*, 5). State 4 in the general LTS is defined by the fluent values

$$\langle \neg SafetyInjection,\ Overridden,\ \neg PressureBelowLow,\ PressureAbovePermit \rangle,$$

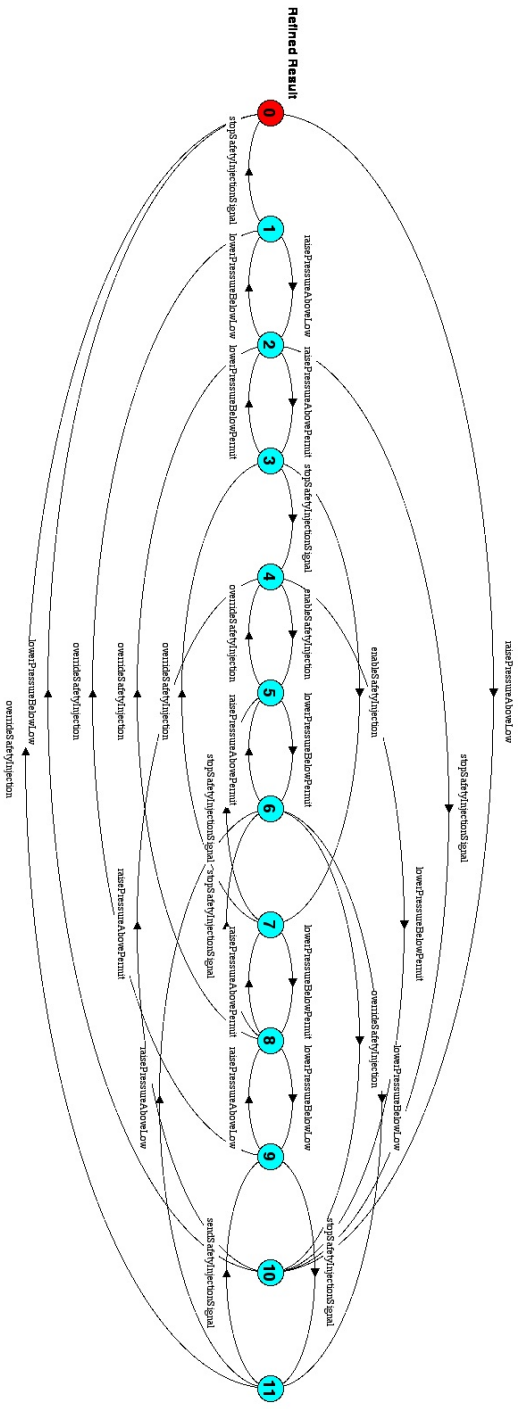

Figure 7.3: Refined Model for ESFAS System

Its outgoing transition matches the outgoing transition from state 2 in the positive LTS, which is *also* defined by the same fluent values. Once again, the *if*-condition at line 10 is *not* satisfied, and so the transition between states 4 and 5 in the system LTS remains in the LTS. As a result, we obtain the refined model in Figure 7.3.

We can compare the results obtained through our approach, to those that the ILP approach would return when applied to the same system. Thus we can comment on the correctness and completeness of our approach. In this case, the resulting model returned by the ILP technique is shown in Figure 7.4.

Our first impressions gathered by just looking at these images would be that the two methodologies return different results. However, to check whether this is true or not, we need to analyse transitions in each of the models. This is because, as mentioned previously, state numbers do *not* play any significant role in the representation of system models, and are merely used as labels for the states. It is therefore possible that our approach numbers states differently to the way the ILP approach does.

Let us try and find a mapping between the states of each model. The results are shown in Table 7.7, which shows the correspondence between states in the model returned by ILP, and those in the model returned by our refinement approach. In addition, we have included a third column which affirms for each pair of equivalent states, whether they are equivalent in terms of their outgoing and incoming transitions. If we can confirm this for each and every pair, then we can conclude that both models are identical.

ILP State	***Refinement State***	***Equivalent?***
0	0	✓
1	10	✓
2	4	✓
3	5	✓
4	6	✓
5	11	✓
6	9	✓
7	8	✓
8	7	✓
9	3	✓
10	2	✓
11	1	✓

Table 7.7: Correspondence between ILP and Refinement states

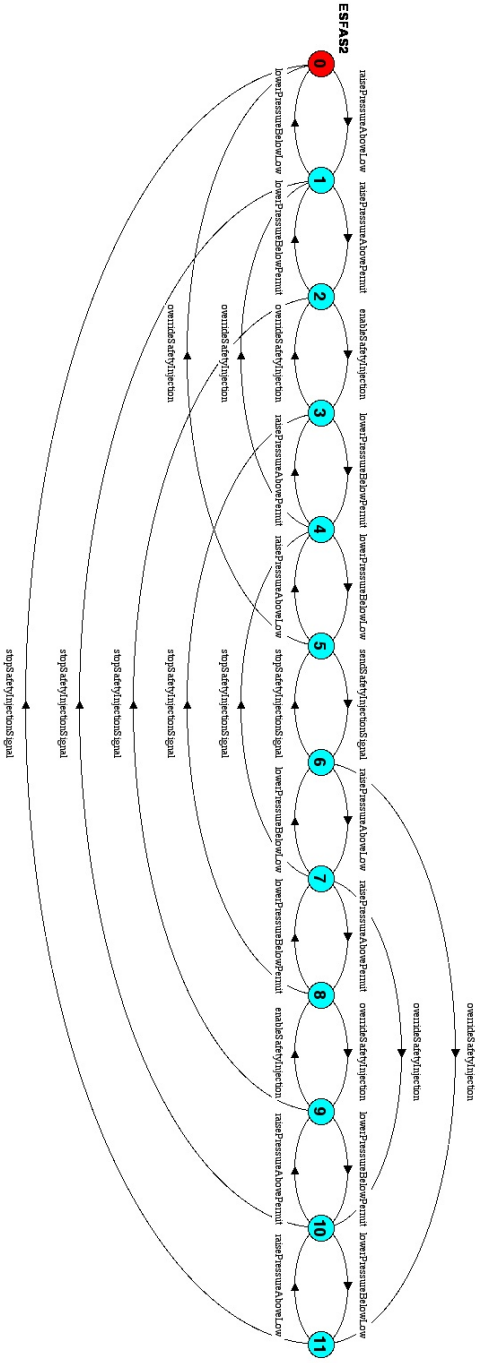

Figure 7.4: Result from ILP approach for ESFAS

As Table 7.7 shows, each and every state in our model is exactly the same as its corresponding state in the ILP model, which proves that both methods return the same resulting LTS for the system in question.

The following chapter assesses the performance of our approach against different criteria, and hence helps determine its usefulness with respect to the achievement of our goals.

Chapter 8

Evaluation

8.1 Discussion

As explained in Chapter 2.1 of this report, we have looked at bottom-up techniques which output the most general LTS corresponding to a given system, so that the resulting model will not accept any new scenarios which exhibit negative behaviour. In our case, we have developed a top-down approach which takes in the most general LTS as input, as well as sets of positive and negative scenarios covered by this LTS, and it outputs a resulting system model that is refined and covers all the positive scenarios.

We have also considered existing techniques that rely on logic programs [2, 3] to learn a set of theories which can be applied to LTSs to synthesise them, as the models accept only those scenarios leading to desirable system behaviour and reject all others. However, these often rely on methods such as language bias that limit the search space when dealing with particularly large and complex systems, by imposing certain syntactic constraints on the hypotheses. By applying our algorithm, we can bypass this form of additional processing carried out by the inductive learning approach, and our methodology can operate directly on the behavioural models without dealing with any declarative properties. However, because we use a non-declarative representation of the behaviour models, it is more difficult to formally prove the correctness of our approach. We have only given an empirical evaluation of its outcomes.

The approach presented in [10] is one of the possible ways in which a subset of a language can be learnt by iteratively partitioning the states of the initial automaton, but its downfall is that it only takes into account the positive scenarios corresponding to a system specification. Conversely, we

consider negative scenarios to be equally as important during the refinement process, as these tend to be quite common in the examples that stakeholders typically provide, and we can think of them as being the safety properties that need to be satisfied in the resulting LTS. Consequently, we use them in our approach as a way of avoiding the problem of over-generalisation that may result from inference rules applied to the positive scenarios.

Another limitation of some of the other algorithms that we have studied and included in Chapter 2 is their restricted application to pure automata. On the other hand, it should be possible to apply our algorithm to systems defined by fluents and events as well.

We can comment on the algorithm's correctness amongst other characteristics, and hence determine the value of its contribution to the process of LTS refinement compared to the current state of the art. Amongst all the different approaches analysed, the ILP technique is the closest to the one we present in this report, as both methods try to explore the generation of behaviour models from scenarios. Hence, the different evaluation criteria used will compare our algorithm to the ILP approach.

Correctness The results of the testing performed in the previous chapter show that our algorithm works on a given LTS and returns a more refined model which is the same as the one obtained by using the ILP approach on the *same* LTS.

However, using our approach can prove to be advantageous for a number of different reasons as mentioned earlier, but the differentiating contribution in this case is that the methodology we have developed during the course of this project works *directly* on the input LTS.

Completeness In contrast to some of the other similar approaches in this field, our algorithm considers both *positive* and *negative* scenarios in order to refine the given LTS. However, it is not possible to supply *all* the possible positive scenarios related to a system as input, as there could be infinitely many. Therefore, we ensure that the resulting model is one which still covers *all* the examples of positive behaviour given as input, whilst rejecting *all* the negative examples provided, without necessarily guaranteeing that the final LTS will meet the given system goals.

Termination The algorithm is guaranteed to terminate, since we assume that the given sets of positive and negative scenarios are covered by the initial LTS, and so any example of undesirable system behaviour expressed

through the negative scenarios is used to prune the corresponding transitions from the general LTS. It may take a sequence of refinement stages to obtain the most specific LTS, but the final stage is eventually reached when the resulting system does not cover any negative behaviour. This will also be the minimal LTS, which encompasses the least number of states. However, it is important to note that the solution produced is not always unique.

This is because each refinement is specific to the set of positive and negative scenarios provided. Therefore, varying these whilst maintaining the same system may cause the solution to vary as well. In other words, for a given LTS to which we apply our refinement algorithm using examples of positive and negative scenarios, the result obtained will be different if we were to use different examples.

Generality All the different examples used throughout this report show that our algorithm can be usefully applied to systems defined by FSPs or domain knowledge alone. In this case the objective was to learn a grammar from given positive and negative examples. In addition, it should be possible to include pure automata defined by strings. Unlike the ILP, we are not restricted to handling system behaviour expressed in the EC only, but it remains to test our approach on systems with varying nature. Hence, the algorithm may allow for the addition of certain extensions which prove its greater usefulness as a flexible mechanism.

8.2 Implementation

We have modified the existing LTSA interface slightly to enable usage of the refinement algorithm developed during this project. However, the primary purpose of this is the demonstration of the concept and functionality, rather than a complete commercial implementation.

As we have shown through the content in Chapter 3, the *refinement* package is highly modular, and can therefore be extended with minimal modifications. The *makeCompactState* method in the main *Refiner* class is all that is needed to translate structures represented using our chosen model representation, to the existing representation used by the LTSA. However, it *may* be necessary to make corresponding changes to the *HPWindow* class in the *ui* package for any alterations in the displayed results, since this is the class that links the existing source code for the LTSA to the new package.

In order to assess the overall performance of our implementation in a holistic manner, it is important to perform an evaluation according to vari-

ous perspectives, as outlined below.

Efficiency We are interested in minimising the number of steps required by our algorithm. A major reduction is achieved by bypassing the additional steps required by the ILP task, such as the translation of the initial specification expressed in LTL to EC, and the use of language bias techniques that constrain the search space. In our case we can differentiate three main stages:

1. Representation of the input scenarios as PTAs
2. Generalisation of the positive PTA
3. Refinement of the given LTS

It is trivial that the number of steps can increase greatly for complex systems with much more extensive alphabets and fluent sets, but this is also greatly dependent on the number and size of scenarios provided. However, to be able to note a considerable change, it would be necessary to study systems that are much larger and complex to the extent that they are hard to understand, and would therefore require a lengthy process of analysis which time constraints would not allow for.

Complexity & Computational Time Apart from the number of steps required by our refinement algorithm, we are also interested in reducing the actual time taken to execute the various steps. The case study from the previous chapter included a system LTS with 12 states, 40 transitions, 4 different fluents, and 9 event labels excluding the hidden action *tau*. Both the positive and negative PTAs corresponding to the input sets of scenarios were composed of 6 transitions and 7 states. In this case there was no need to generalise the positive behaviour entailed by the positive scenarios, so the positive PTA was used during refinement. As a result of the refinement, the number of states in the original system LTS remained unaltered, but the number of transitions was reduced to 30.

In contrast to the ILP approach, our algorithm does not need to perform an exhaustive search on the input, and can bypass the translation process of LTL into EC as well as the language and search bias phases applied to restrict the hypotheses. Therefore, we would expect improvements in the speed with which results are returned by our methodology compared to the time taken by the inductive learning approach. However as mentioned

previously, formal verification of results is much more difficult with our approach due to the lack of any declarative representation.

In the case of the case study presented in the previous chapter, as well as for all the other examples mentioned in this report, the LTSA returned results from both the generalisation and the final refinement in a matter of seconds. Though we have not calculated the exact time taken, we have realised that results are computed and displayed in the LTSA *Draw* tab within a time that appears as immediate once the user chooses the *Refinement* $\longrightarrow$ *Refine* option.

Execution time could also increase with the number of steps involved in the various stages of our refinement algorithm. For instance, the provision of dense sets of scenarios would lead to a more time-consuming construction of the corresponding PTAs, and generalisation of the positive PTA, as a greater number of states and transitions would need to be taken into account. In addition, if the given LTS itself includes numerous states and transitions, then it would take more time to perform the refinement process on it due to the extra processing required.

Chapter 9

Conclusions and Future Work

9.1 Discussion

This report describes a methodology for refining LTSs generated from FSP descriptions, in accordance with sets of user-provided scenarios. The approach undergoes two different stages, the first of which aims to generalise system behaviour in order to preserve as much of the desirable properties of the system as possible, whilst the second stage uses this generalised model together with examples of undesirable system executions in order to output a refined LTS which encompasses all the positive behaviour and simultaneously disallows any negative one.

A prototype has been developed as part of the existing LTSA tool, which shows the algorithm in successful operation as it returns the desired output, but there is room for further work. The tool has been tested on several (non-) trivial examples from the literature, and despite the appropriateness/suitability of results obtained when compared to the results from the ILP on the same system, there are other areas which could be looked into in order to enhance the current functionality provided by this tool, thus increasing the value of its contribution to the requirements engineering domain. These are discussed in the next section.

9.2 Additional features and future work

The current version of the tool satisfies our initial requirements of being able to generalise positive system behaviour, and using the resulting model

together with examples of undesirable behaviour, to generate a modified system model that preserves the desirable behaviour whilst rejecting any occurrences of unwanted system execution. However, there are a number of different features that could be included within our existing tool. These are discussed in more detail in this section.

As discussed in Chapter 4, our lattice of automata shows that the numbering of states influences the order in which state-merging is performed during the generalisation of the positive PTA, but as our current algorithm stands, it will not always return the smallest generalised model in terms of the number of states and transitions in the result. To guarantee minimality in the positive LTS, the necessary code can be added to the generalisation part of the algorithm, which in addition should go through all the different merging options available for a set of states. Consequently it should store all the different structures in order to eventually return the one which contains the least number of states.

One enhancement of the existing tool could be to include LTSs of a higher complexity for the refinement process, such as those including tau actions, 'tick' events, and so on. During the course of this project we have assumed that all input LTSs are defined by sequences of events between system states, without taking into account any notion of time. By including these in our study we could benefit from the ability to cover a vast array of systems, and hence provide a more general approach that considers these differences.

Throughout this project we have assumed that everything starts at the same unique initial state. However, if fragments of scenarios are provided by different users, then additional checks will be needed to ensure consistency of the scenarios.

Often in larger systems, an FSP file contains the definition for a number of sub-processes, and the final process is just a result of composing these individual processes. If, however, we want to simultaneously perform the refinement on two different LTSs that are closely linked, it would be a reasonable idea to extend our algorithm to cover this genre of situations. We would need to take into account various different issues such as shared actions, concurrency issues, and so on.

At the present stage, we are also assuming that scenarios are complete, i.e. that the events appearing in these scenarios are the only events possible in the system. However, we may want to relax these assumptions and learn requirements from a set of incomplete scenarios that satisfy a particular system specification. This would help cover implied scenarios in addition to the ones that we have been dealing with during this project.

Given that the principal reason for the existence of our tool is to satisfy end-user requirements fully, and to not cause or allow any form of system behaviour that would be classified as unacceptable by the user, it would be an added value to the users if we could provide them with different ways of generating the input scenarios. Currently we assume that the scenarios are already contained within text files that users have access to, and which they can select for a given LTS that needs to be refined. A possible enhancement could be an interactive version of our approach, whereby users can actually create scenarios on the fly, and feed these into our system using a GUI, so that the generalisation algorithm can take these into account and adjust its output accordingly. The dynamic addition of scenarios using MSCs as in [34] would be helpful to verify how easily the refined LTSs can be adapted, and their tendency to errors.

Scenarios could also be generated automatically, and so it remains to investigate the integration of our refinement approach and model checking techniques in order to find new ways to increase the flexibility and efficiency of our approach.

Since the FSP files that we have been working with in order to test our implementation have just included domain conditions and no further specification of goals, triggers, and so on, further research would be necessary to look into the inclusion of such pieces of information, including post-conditions, which on the other hand state properties that should *not* hold as a result of a certain event. An addition to this would be to include user-defined goals.

We can think of our current refinement algorithm as being a means through which we can satisfy certain safety properties of the system. It would be interesting to adapt the current approach so that it can also handle other types of system properties, such as liveness, fairness, and timed properties.

Bibliography

[1] *http://www.eti.pg.gda.pl/katedry/kiw/pracownicy/jan.daciuk/personal/thesis/node12.html*.

[2] D. Alrajeh, O. Ray, A. Russo, and S. Uchitel, *Using abduction and induction for operational requirements elaboration*, in Journal of Applied Logic, 2008.

[3] D. Alrajeh, A. Russo, and S. Uchitel, *Extracting requirements from scenarios with ilp*, 16th International Conference on Inductive Logic Programming, (2006).

[4] D. Angluin, *Inference of reversible languages*, J. ACM, 29 (1982), pp. 741–765.

[5] D. Bošnački, S. Leue, and A. L. Lafuente, *Partial-order reduction for general state exploring algorithms*, in SPIN, 2006, pp. 271–287.

[6] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu, *Learning assumptions for compositional verification*, 2003.

[7] P. Courtois and D. L. Parnas, *Documentation for safety critical software*, in Proc. of 15th ICSE, 1993, pp. 315–323.

[8] C. Damas, P. Dupont, B. Lambeau, and A. van Lamsweerde, *Generating annotated behaviour models from end-user scenarios*, IEEE Transactions on Software Engineering, (2005).

[9] C. Damas, B. Lambeau, and A. van Lamsweerde, *Scenarios, goals, and state machines: a win-win partnership for model synthesis*, IEEE Transactions on Software Engineering, (2005).

[10] T. Elomaa, *Partition-refining algorithms for learning finite state automata*, in ISMIS, M.-S. Hacid, Z. W. Ras, D. A. Zighed, and Y. Ko-

dratoff, eds., vol. 2366 of Lecture Notes in Computer Science, Springer, 2002, pp. 232–243.

[11] D. Giannakopoulou and J. Magee, *Fluent model checking for event-based systems*, in Proceedings of FSE, ACM Press, 2003.

[12] J. E. Hopcraft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley, 2001.

[13] M. Huth and M. D. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, 2000.

[14] H. Ichikawa, M. Itoh, J. Kato, A. Takura, and M. Shibasaki, *Sde: Incremental specification and development of communications software*, IEEE Trans. Computers, 40 (1991), pp. 553–561.

[15] A. Kakas, R. Kowalski, and F. Toni, *Abductive logic programming*, Journal of Logic and Computation, 2 (1992), pp. 719–770.

[16] J.-P. Katoen, *Labelled transition systems*, in Model-Based Testing of Reactive Systems, vol. 3472 of Lecture Notes in Computer Science, Springer, 2005, pp. 615–616.

[17] R. Kowalski and M. Sergot, *A logic-based calculus of events*, New Generation Computing, 4 (1986), pp. 67–95.

[18] J. Kramer, J. Magee, and M. Sloman, *Conic: An integrated approach to distributed computer control systems*, in IEE Proc., 1983, pp. 1–10.

[19] I. Kruger, R. Grosu, P. Scholz, and M. Broy, *From mscs to statecharts*, in Int'l Workshop Distributed and Parallel Embedded Systems, 1998, pp. 61–72.

[20] A. V. Lamsweerde, *Goal-oriented requirements engineering: A guided tour*, in Proc. of 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2001.

[21] A. V. Lamsweerde and L. Willemet, *Inferring declarative requirements specifications from operational scenarios*, IEEE Trans. on Software Engineering, 24 (1998), pp. 1089–1114.

[22] E. Letier, *Goal-oriented elaboration of requirements for a safety injection control system*, tech. report, Dèpartement d'Ingènierie Informatique, UCL, 2002.

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

[24] E. Mäkinen and T. Systä, *Mas - an interactive synthesizer to support behavioral modelling in uml*, in Proc. ICSE 2001 - Int'l Conf. Software Eng., IEEE Computer Society, 2001, pp. 15–24.

[25] R. Miller and M. Shanahan, *The event calculus in classical logic - alternative axiomatisations*, Linkoping Electronic Articles in Computer and Information Science, 4 (1999), pp. 1–27.

[26] R. Miller and M. Shanahan, *Some alternative formulations of event calculus*, in Computational Logic: Logic programming and Beyond, vol. 2408 of Lecture Notes in Computer Science, Springer, 2002, pp. 452–490.

[27] S. H. Muggleton, *Inductive acquisition of expert knowledge*, PhD thesis, 1986.

[28] R. D. Nicola and F. W. Vaandrager, *Three logics for branching bisimulation*, Journal of the ACM, Vol. 42 (1995), pp. 458–487.

[29] J. Oncina and P. Garca, *Inferring regular languages in polynomial update time*, World Scientific Publishing, 1992, pp. 49–61.

[30] O. Ray, *Hybrid Abductive-Inductive Learning*, PhD thesis, Imperial College London, 2005.

[31] O. Ray, *Using abduction for induction of normal logic programs*, in ECAI'06 Workshop on Abduction and Induction in Artificial Intelligence and Scientific Modelling, P. Flach, A. Kakas, L. Magnani, and O. Ray, eds., 2006, pp. 28–31.

[32] E. Sandewall, *Features and fluents: The representation of knowledge about dynamical systems*, Oxford University Press, (1994).

[33] C. Stirling, *Temporal Logics in Specification*, 1987, pp. 1–20.

[34] S. Uchitel, J. Kramer, and J. Magee, *Synthesis of behavioral models from scenarios*, IEEE Trans. Software Eng., 29 (2003), pp. 99–115.

[35] A. van Lamsweerde and L. Willemet, *Inferring declarative requirements specifications from operational scenarios*, IEEE Trans. Software Eng., 24 (1998).

[36] J. Whittle and J. Schumann, *Generating statechart designs from scenarios*, in ICSE 2000: Proceedings of the 22nd International Conference on Software Engineering, ACM Press, 2000, pp. 314–323.